11-1-2011

# An Analysis of open security issues of Android interfaces to cloud computing platforms

Corey Beres

# An Analysis of Open Security Issues of Android Interfaces to Cloud Computing Platforms

by

**Corey Andrew Beres**

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering

Supervised by

Associate Professor Dr. Shanchieh Jay Yang
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
November 2011

Approved by:

---

Dr. Shanchieh Jay Yang, Associate Professor
*Thesis Advisor, Department of Computer Engineering*

---

Dr. Andres Kwasinski, Assistant Professor
*Committee Member, Department of Computer Engineering*

---

Dr. Roy Melton, Senior Lecturer
*Committee Member, Department of Computer Engineering*

# Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

An Analysis of Open Security Issues of Android Interfaces to Cloud Computing Platforms

I, Corey Andrew Beres, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

---

Corey Andrew Beres

---

Date

# Acknowledgments

I would like to thank my parents who drove me to and from school too many times. Without their support none of this would have been possible. I am grateful for the help of Dr. Yang, whose assistance was immeasurable, and for my lab mates, Jon, Daniel, and Puey Wei.

# Abstract

**An Analysis of Open Security Issues of Android Interfaces to Cloud Computing Platforms**

**Corey Andrew Beres**

**Supervising Professor: Dr. Shanchieh Jay Yang**

Smartphone usage is on the rise and some may argue that these devices are ubiquitous in today's society, even among non-technical users. To remain competitive, mobile devices and applications need to quickly perform tasks with as minimal as possible impact on battery life. The emergence of cloud computing, open-source cloud platforms, and cloud-supported ventures such as Apple iCloud and Amazon Silk provide new and promising methods to improve device and application performance. However, little work has been done to examine the security of offloading processing from mobile devices to cloud services and the performance effects of implementing security features. This work aims to answer the questions that arise in securing mobile applications that communicate with the cloud.

Via a proof-of-concept application that offloaded resource-intensive computations to an open-source cloud computing platform, the security of cloud computing and Android was studied. It was found that, by following recommended coding practices, the cloud-smartphone security landscape could be significantly improved. Further security enhancements were also recommended and summarized. Additionally, performance was analyzed, and it was found that mobile device applications benefit heavily from cloud support and that features such as secure authentication and encryption do not noticeably impact application performance.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Today's IT industry is witnessing the unfolding of a very interesting scenario. Public cloud usage is on the rise; in November 2006, merely 1,000 Amazon EC2 instances were launched per day. By September 2009, the number was estimated to have risen to over 40,000 instances launched per day [1]. This trend is shown in Figure 1.1. Academic interest in cloud computing has increased exponentially as well, as shown in Figure 1.2. In addition to the growing acceptance of public cloud computing, the release of open-source private cloud computing platforms such as Eucalyptus and Nimbus have created the possibility of installing a cloud even in a home.

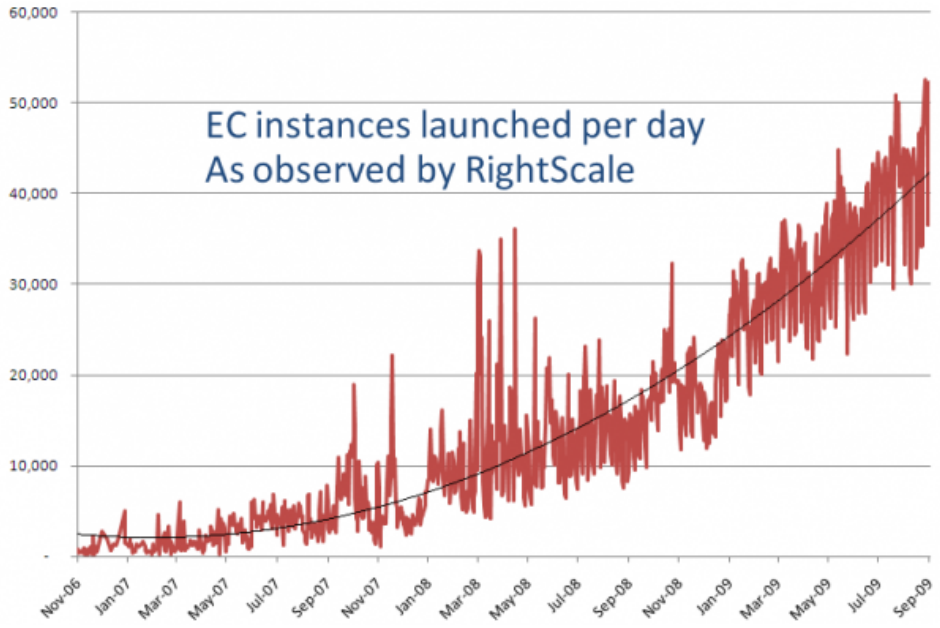


Figure 1.1: Amazon EC2 instances launched per day as observed by RightScale [1]

At the same time, smartphone usage is increasing, and these devices are becoming ubiquitous, even among users without technical backgrounds. For instance, [2] found that smartphone adoption in the United States rose from 17 % to 27 % over the course of 2010.

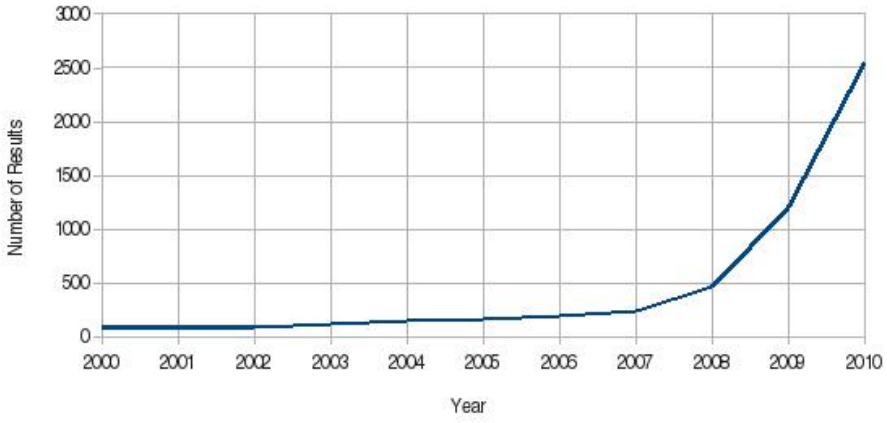


Figure 1.2: Academic interest in cloud computing according to the number of search results for "cloud computing" on IEEE Xplore

In some European countries, the numbers are much higher. Spain's adoption rate rose to 38 %, up from 27 % the previous year. These trends are shown in Figure 1.3.

As smartphone usage grows, the resources available to such devices remain limited. Even as battery technology improves, clock speeds rise, and applications access the Internet for new purposes, limiting the amount of time a device can be used without external power. Conveniently, cloud computing is becoming increasingly available. The technology and motivation exist to offload work from smartphones with limited resources to the energy- and processor-rich cloud.

Several existing mobile applications take advantage of the cloud. Microsoft's Project Hawaii [5] is a software development kit targeted to students that allows developers to leverage services including a relay service, a rendezvous service, optical character recognition, and speech-to-text in the cloud. Access to Microsoft's Windows Azure cloud platform is also supported. Currently the project supports only Windows devices.

In May 2011, Apple unveiled its own cloud venture, iCloud [6]. The service allows users to store data in the cloud, including music, photographs, documents, books, and contacts. The data can be retrieved by supported Apple devices.

Popular Android applications like Google Goggles [7] offload work to the cloud. Goggles is a visual search application created by Google that allows users to take photographs of objects to find more information on those objects. The application works by sending the photos to the cloud, where they are processed on Google's servers. The cloud then returns search results to users' mobile devices. Another Android application that leverages the cloud is Amazon Silk [8], a web browser created by Amazon and available on Kindle devices that offloads processing and caching to Amazon EC2.

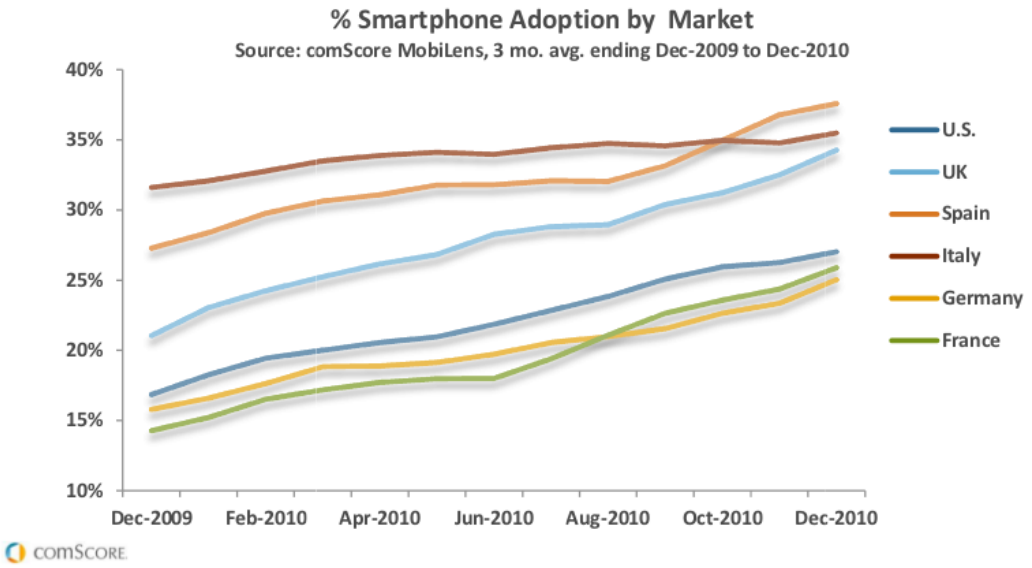


Figure 1.3: Smartphone adoption by market [2]

However, there are many areas for vulnerabilities to develop when a smartphone and a cloud interact. These include storage of possibly secure data once it arrives on mobile devices, storage of possibly secure data on the cloud, and secure transmission of data between the cloud and devices. Additionally, there are many instances of malware that threaten security of mobile devices. Figure 1.4 shows the growth in malware on mobile platforms from 2009 to 2011 (over which time the amount of malware has doubled) as recorded by McAfee [3]. Figure 1.5 shows the breakdown of malware by mobile platform; Android was a target for the majority of malware considered in the survey. Additionally, there have been numerous instances of impropoper management of private data by applications as in [9] and [10].

The scenario of cloud-smartphone interaction opens many uncertainties, such as the vulnerabilities that arise when a smartphone and a cloud commingle. Very little research exists on this topic, and the specific vulnerabilities have not been concretely defined. As such, no answers have been offered as to whether security needs to be improved and what specific security features need to be added and expanded. This thesis is mainly concerned with the Android operating system.

In an effort to clarify those uncertainties, CloudSmart was created. This software system tied together an Android smartphone with a Eucalyptus private cloud. The system provided an interface on the smartphone, offloaded work to a cloud, and assembled and returned the results from the cloud to the smartphone. A specific situation in which the cloud would be very advantageous was imagined, and it proceeds as follows:

1. Ray trace a scene.
2. Retrieve images from the Web or a local cache.


Figure 1.4: Growth in malware on mobile platforms [3]

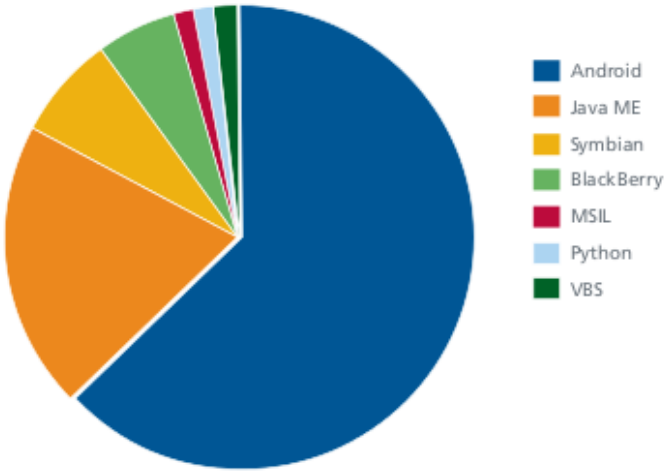


Figure 1.5: Breakdown of malware by mobile platforms, Q2 2011 [3]

3. Compare the ray-traced scene to the images and find the closest match.

Through this work, common and possible security practices that posed security risks to Android devices, the cloud, and users were identified. Similarly, vulnerabilities in the applications were identified. Vulnerabilities that posed substantial risks were identified, allowing them to be addressed with future research. For those goals, a stronger focus was placed on the Android end of the spectrum rather than the cloud end. The work presented here is not intended to be a comprehensive security solution.

With risks identified, methods of exploitation described, and solutions proposed, the results of the work will facilitate the development to better and more comprehensive Android and cloud security models. Proposed solutions can be implemented and improved, and the results will exist for future researchers interested in the critical areas that might pose security threats.

Chapter 2 includes a discussion of past work related to this project. In Chapter 3, the specific application and its implementation are explained, including a list of security features and practices that were found to be important. Chapter 4 contains performance results from the application developed along with explanations of how the results were found.

# Chapter 2

# Related Work

The concept of a cloud-computing interface for a smartphone is relatively new. Amazon EC2 and Eucalyptus were officially released in 2008, while the Amazon Web Services SDK for Android was unveiled at the end of 2010. As of October 2011, Eucalyptus lacks an Android interface. However, there are some projects in which the security of Android and the cloud was addressed.

Two similar projects with intentions to secure the Android landscape are Saint [11] and Kirin [12]. According to its authors, Saint is a framework for developers that allows for "applications to protect themselves." This goal was accomplished by adding developer-defined installation-time and run-time policies. A main benefit of Saint is that it can add dynamic protection for applications, whereas most built-in security features provide static behavior. The installation-time protection included signature-based policies and configuration-based policies. The former allows applications that declared permissions to grant their permissions based on signatures of requesting applications, while the latter performs the same feat based on the requesting applications' configurations (*e.g.*the permissions requested and its version). At run-time, the previous two policies are in effect along with a context-based policy, which limits the inter-process communication (IPC) between two applications based on the state of the phone, (*i.e.*what hardware is active). At run-time, a signature-based policy would limit IPC based on the signatures of applications, while a configuration-based policy would limit IPC based on the configurations of the applications in question. For run-time protection, Saint acts as a middle-man for IPC.

According to the authors, "Saint was implemented as a modification to the Android 1.5 OS." Requiring modifications to Android is the main limiting factor to the adoption and usefulness of Saint. Also, the Saint package installer requires Android's *PackageParser* class, which is hidden. In addition, the application would most likely require the INSTALL_PACKAGES permission, which is not available to applications installed from the Android Market. Even if Saint were added to the Android OS, the additional restrictions would be up to developers to define; thus its use would be optional.

Kirin is another framework for Android. Its developers focused on adding policies that are checked during the installation of new applications. These fell into three categories: regular Android permissions, mutual exclusion of permissions (permissions that cannot be held simultaneously by an application), and rights dependence (permissions that must be held if another permission is held). Specifically, the policies introduced included:

- Explicit permission must be granted to make voice calls.
- Applications that were granted dangerous permissions much have no unprotected components.
- Only system applications can interface with hardware and process outgoing calls.
- Applications that can record audio cannot have Internet access and cannot pass data to applications that have Internet access.
- Applications that can access WiFi or a device's network state must have the Internet permission.
- Applications can receive SMS updates and location updates only from trusted system components.

Kirin addressed several deficiencies present in Android, and, combined with compulsory and wise use of Saint, it would lead to a more secure Android environment. However, Kirin suffers from the same downfalls of Saint: notably the requirement of classes hidden from the SDK. Additionally, the authors of Kirin and Saint did not specifically address cloud interaction.

The creators of [13] aimed to identify when a device was compromised as quickly as possible while maintaining the efficiency of this detection. The project was named Paranoid Android, and it was accomplished by emulating Android devices in the cloud. The authors' example implementation detected intrusions with tamper-evident secure storage, detection of software errors in Android, memory analysis, open-source antivirus software, and methods to scan memory for malicious code, but they maintained that any check could be implemented, with the detection time proportional to the number of checks performed. While the project's goals differ from this project's goals, its usefulness in monitoring the security of smartphones connected to the cloud is intriguing.

A project with similar goals to this project but with a different means and focus is described in [14]. The goal of CloneCloud was to offload work automatically for any application from smartphones to the cloud. This was accomplished by a method similar to that of Paranoid Android. Devices were emulated and "cloned" to run in the cloud. Threads of execution were automatically migrated by the CloneCloud framework to the cloud, where they executed and took advantage of the hardware of much more powerful servers until the threads were merged back into the original process. This project was based on an early version of the Android mobile operating system.

Most of the project's work involves the decisions of when to offload work, how to send threads to the device clones on the cloud, and how to return the threads to mobile devices. The end results of the application included a speed-up of execution time of up to 20 times and up to a 20-times decrease in battery usage. While the results are very promising, the focus of the project was application performance, and security was not a concern. The project also required modifications to the Android operating system that could not be easily rolled out on-demand to any smartphone.

## 2.1 Cloud security

Cloud security has been studied in many projects, including [15] in which the evolution and modern applications of security features were explained, and [16] in which modern security features like WS-Security and Transport Layer Security (TLS) were explained. As described in those projects, security in the cloud includes those features found in the desktop and server realms. Users are isolated from each other and have limited permissions. Virtual machines isolate running software and memory within a physical machine. A more interesting case results when protecting access to a cloud.

WS-Security, short for Web Services Security, is an extension to SOAP that defines "how to provide integrity, confidentiality and authentication for SOAP messages" [16]. SOAP, or Simple Object Access Protocol, is a protocol that defines how to transfer information between Web services. It relies on the XML standards XML Signature and XML Encryption, and it defines the X.509 certificate. XML Signature defines how XML markup can be signed to authenticate messages. Very simplified, the standard works by hashing message parts, signing an element, and storing the result of the signature in another element, which is included in the security header. XML Encryption allows for XML markup to be encrypted. The encrypted markup is separated into two elements, the encrypted data and the key used for the encryption, which is also encrypted, this time using the public key of the recipient.

Transport Layer Security consists of two parts: the Record Layer encrypts data, and the Handshake is responsible for authenticating servers and clients. The most common usage of TLS begins with a server outfitted with an X.509 certificate issued from a trusted certificate authority. During the handshake, the client receives the certificate from the server and verifies its contents. If the certificate cannot be verified, the user is prompted to accept or reject the certificate. The vulnerabilities begin here, as users may quickly click "Accept" and "Okay" to reach their desired content without understanding the risks that were taken.

Attacks to circumvent these cloud security mechanisms exist, and they are described in [16]. XML Signature Element Wrapping involves stealing a signed SOAP message body and placing it inside the SOAP header of a new message. The body of the new message describes the action the attacker wants to perform. This allows an attacker to perform his own actions with the credentials of a different, legitimate user. The attack was used in 2008 to exploit Amazon EC2. Another attack is a metadata spoofing attack in which a Web Service's metadata is altered, resulting in unintended execution of commands different from the commands they intended to execute. This attack can happen after a user happens upon this invalid metadata. A final attack comes in the form of the classic denial of service attack, which, due to the elastic nature of the cloud, could potentially bring down an entire cloud as exponentially more resources are provisioned to endure the attack.

Another security concern is the main interface to the cloud: the Web browser. Web browsers do not take advantage of XML Security and XML Encryption. Rather, they use the encryption features of TLS only, and the signature is used only during the handshake. Additionally, the Same Origin Policy (SOP), which browsers use to identify origins of tokens and scripts, can be duped by invalid and outdated DNS cache data. The authors of [16] concluded that authentication via browsers is insecure because Web browsers cannot

"issue XML based security tokens," and security tokens are stored within the browser where they are protected only by SOP.

Additional issues described in [15] include the lack of reputation fate-sharing, a lack of mutual auditability, the possibility that administrators are seen as "decision bottlenecks" as they enforce tighter restrictions upon users who seek immediate access, and side- and covert-channel attacks as potentially competing entities run their software on possibly the same machine within the cloud. The previously described vulnerabilities are summarized in Table 2.1.

## 2.2 Android security

As described in [17] and [18], the security situation with Android is similar to that of cloud computing in that a number of security features are inherited from the desktop realm along with several additional attributes that tighten the security model. These mechanisms are summarized in Table 2.2. Features found in desktops include POSIX users, file permissions, virtual machines, and the Java language. The first three isolate applications from each other. POSIX users have been adapted to Android such that each application runs as its own user; thus it can save files that only the application can access, and it cannot access the private files of other applications. Virtual machines have also been adapted to Android: each application runs in its own virtual machine in order to isolate memory . Finally, the Android SDK supports only Java, which forces users to take advantage of Java's type safety and memory management that prevent memory-level attacks.

The most prominent security feature unique to Android is the permissions system [19]. Permissions are defined such that dangerous mechanisms within Android are protected, including hardware resources like Bluetooth, WiFi, the phone, microphone, and external storage. Applications must have the necessary permissions granted to them when they access such resources. These permissions are granted by the user upon installation of applications, and they cannot change. An application cannot be installed if any permissions are rejected. Finally, applications can also be protected with permissions. An application can require that the applications that launch it hold certain permissions. The benefits of the permissions system are that users know what critical resources an application will access and users can identify dangerous combinations of resources.

Additional Android-specific security features include the following: component encapsulation, which allows components of applications to be declared as private, thus preventing their access by other applications; application signing, which identifies the developers of applications; mobile carrier security features, which authenticate phone users with SIM cards; and Intent filters. Intents [20] are a feature of the Android SDK, and they are used when an application attempts to start another application (allowing the source application to identify what application it would like to start, either by name or by attributes of possible matches). Intent filters limit the scenarios in which applications can be launched, and they can be defined on a per-component basis. Application components are the main processes of these applications; the most common components are activities (foreground processes that provide a visible user interface) and services (background processes).

Table 2.1: Summary of cloud vulnerabilities

| *Vulnerability* | *Consequences* |
|---|---|
| XML Signature Element Wrapping: including a valid signature in an otherwise invalid SOAP message. | In 2008, an attacker performed "arbitrary EC2 operations" with a legitimate user's credentials |
| Web browsers lack support for XML Signature and XML Encryption. | Credentials can be moved from one machine to another to appear as a different user. |
| Web browsers rely on same-origin policy (SOP). SOP can be duped via DNS cache poisoning. | Browsers will allow scripts to access files and run other scripts they would not otherwise be able to access. Browsers will accept invalid tokens. |
| Browsers cannot generate valid XML tokens. | Tokens can be moved from one computer to another, where the only defense is the SOP. |
| Cloud Malware Injection Attack: attacker creates a malicious VM, adds it to the cloud system, tricks the system into thinking it's a valid VM, and finally executes code with the instance. | Code can be executed by a malicious VM. It may not have the same restrictions as valid VMs. |
| Metadata Spoofing Attack: attacker changes metadata sent from the cloud; attacker can change a service's WSDL so that calls to one command look like a different command. | Users can unknowingly execute commands. |
| Denial of Service attack | Cloud will bring up more VMs, which might slow down other VMs until they crash. |
| Reputation fate-sharing and lack of mutual auditability | Business reputations could be damaged because users or providers act maliciously. |
| Administrators becoming "decision bottlenecks" | Users will try to bypass administrative restrictions. |
| Side- and covert-channel attacks | Competing entities could steal intellectual property from each other. |

Table 2.2: Android security features

| *Feature* | *Install-Time Effect* | *Run-Time Effect* | Required? |
|---|---|---|---|
| Android Permissions | Notifies users of dangerous features that will be accessed. Indicates dangerous combinations of conditions. | Protects access to dangerous resources by requiring permissions be held by hardware-requesting applications. | Required to access resources. Optional to protect application components. |
| Intent Filters | *None* | Limits scenarios in which components can be launched. | Optional to protect application components. |
| Component Encapsulation | *None* | Declaring components as private protects them from unintended access by other applications. | Optional |
| POSIX Users | *None* | Each application runs as a different user, isolating applications. | Required |
| File permissions | *None* | Each application has its own private directory. | Using the private directory is optional. |
| Java language features | *None* | Type safety and memory management prevent memory-level attacks. | Required in SDK. |
| Virtual machine | *None* | Each application runs in its own VM. | Required |
| Application Signing | Unsigned applications are not allowed in Android Market. Signature-level permissions are granted only to applications created by the same developer. | Applications with the same signature can run as the same user ID. | Required |
| Mobile carrier security features | *None* | SIM cards are used to identify and authenticate users, preventing call theft. | Required if calls will be made. |

The interplay of various security features can be difficult to understand. Figure 2.1 shows the process of decisions that Android makes when one attempts to start a specific application. Figure 2.2 shows the decisions made when an application attempts to start any application that matches specified criteria. Finally, Figure 2.3 shows the process of resolution when an application attempts to open or save a file. The information in the figures was not gathered from review of actual Android source code. Rather, it was gained through understanding documentation in the Android SDK; thus the order in which the individual checks are presented may not reflect the actual order in which Android performs the checks.

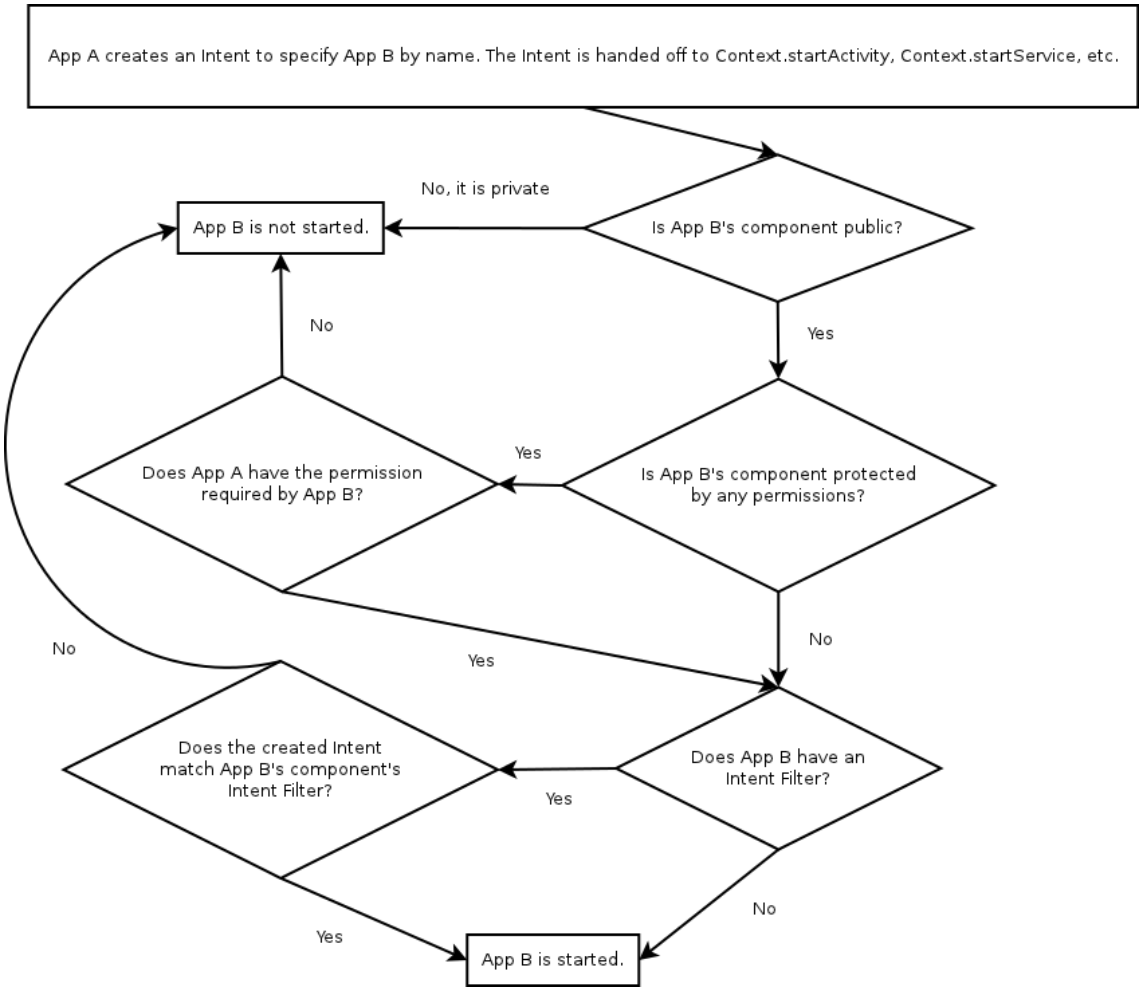


Figure 2.1: Security decisions encountered if application A attempts to start application B

Of course, there are means to circumvent these security features, and a number of them focus on the permissions system. One flaw was found in which protection of an application with a permission did not protect its components [21]. The authors of [22] found that permissions that were declared and then granted were not revoked once the original

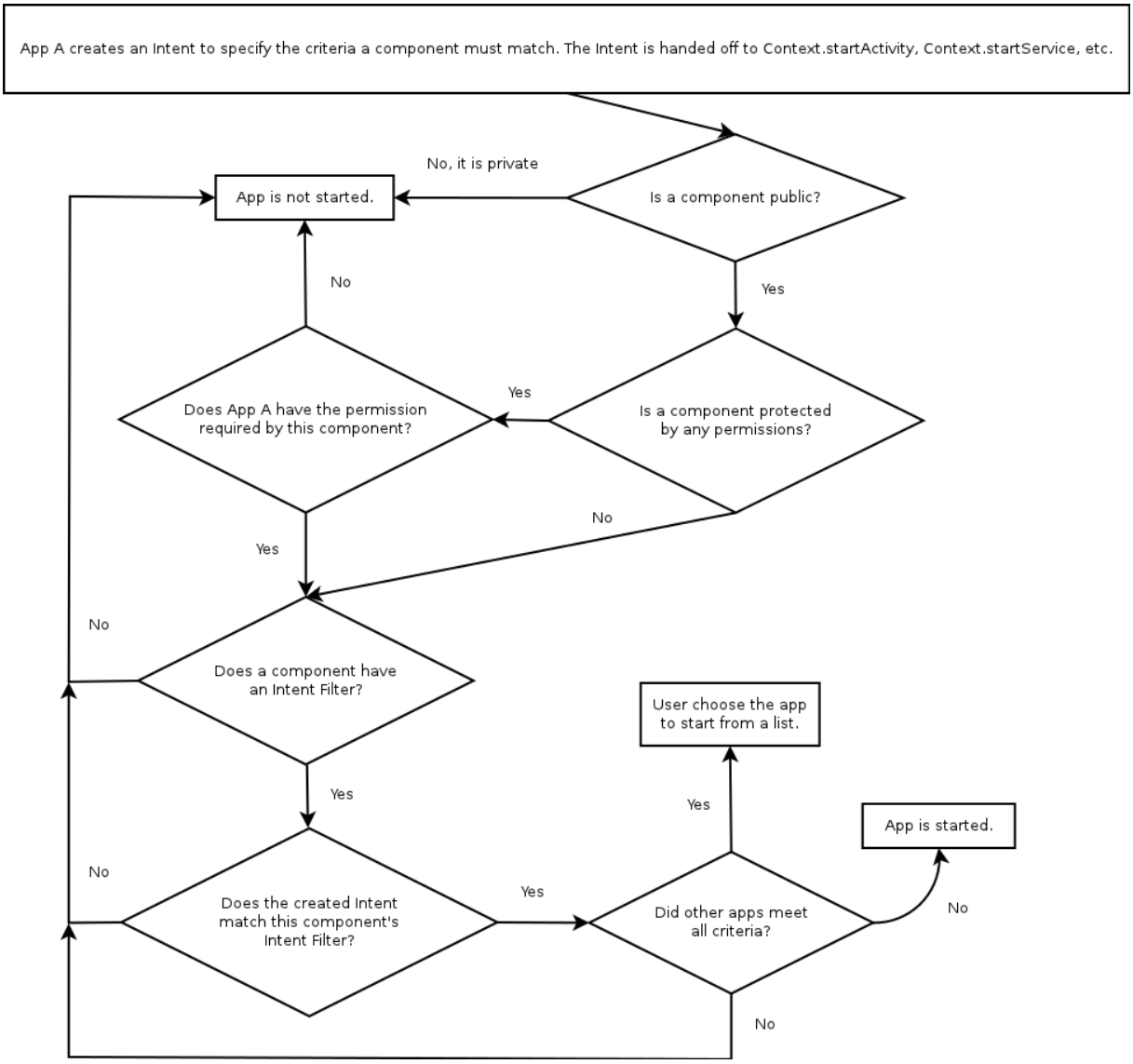


Figure 2.2: Security decisions encountered if application A attempts to start any application that matches specified criteria

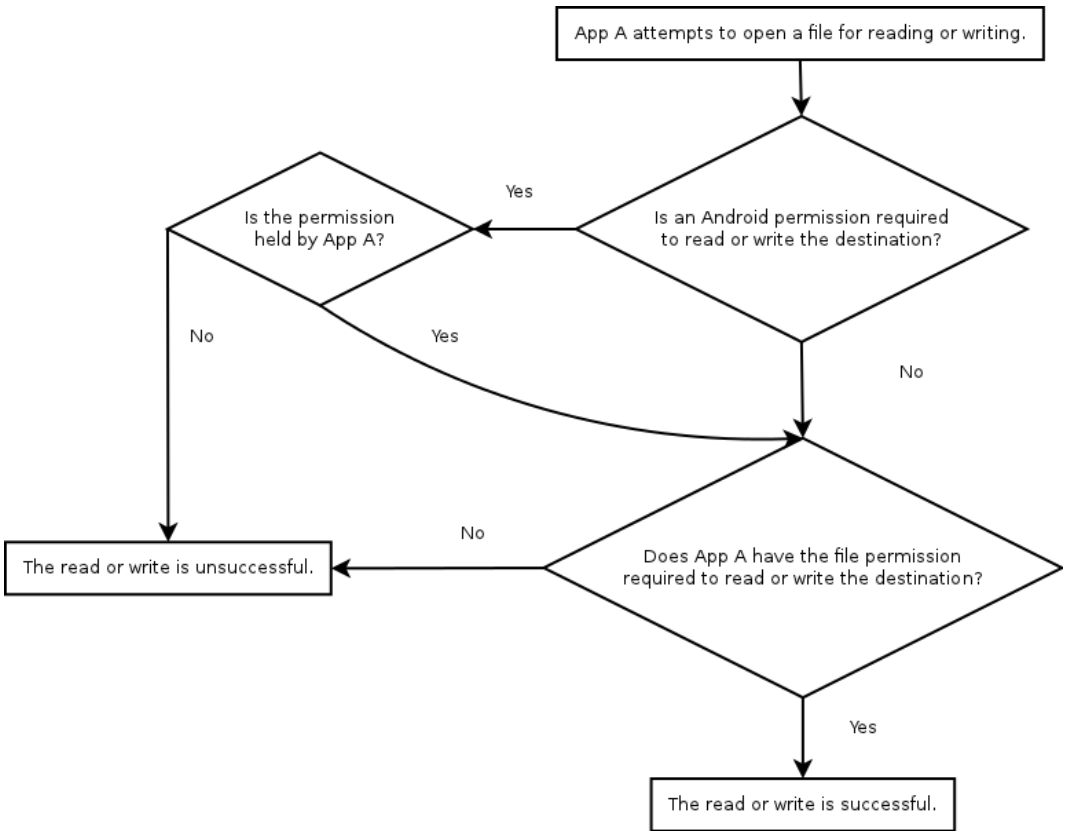


Figure 2.3: Security decisions encountered if application A attempts to open or save a file

declaring application was removed. The latter allowed applications to gain permissions to new applications that were installed later if these new applications declared a permission with the same name as the previously deleted permission. These two flaws have since been addressed in Android. The authors also noted the inclusion of arbitrary levels of permission to denote critical nature (*e.g.*"normal" compared to "dangerous"), that allow certain permissions to be hidden from the user upon installation.

Furthermore, applications that are granted dangerous permissions do not need to be protected by those same permissions, which lets developers create backdoors that can be opened by other applications. For instance, a malicious developer could create an audio recording application that has permissions to record audio and write to external storage, and that is not protected by either of those permissions. A service could be added to the application that records audio and saves it in a public location with no user notification. A separate application with permission to access the Internet could launch the service to record audio and then upload resulting files to the Internet, again with no user notification. This flaw was addressed by [12], but only by modifying the Android operating system. The backdoor could be unnecessary as any application with the Internet permission can upload existing public files to a remote server while the user remains completely unaware.

Defects have also been discovered with the Java Native Interface, such as the vulnerability described in [17] that allowed for binary executables to be included with applications. These executables could execute dangerous actions on the behalf of unprivileged applications. The aforementioned vulnerabilities are summarized in Table 2.3.

Table 2.3: Summary of Android vulnerabilities

| *Vulnerability* | *Consequences* |
|---|---|
| Include malicious binary as a resource, execute it via JNI using Android Build System. | System files can be read. On "rooted" devices, ARM instructions and C programs can be executed. |
| Protecting entire application with permissions does not protect its components. The necessary checks are left to the developer. This can be exploited if memory overflows are possible and Android Scripting Environment are present. | Applications with less permissions can access components of a more privileged application. |
| Permissions are identified by their names. Granted permissions were not revoked when the permissions were deleted, which resulted in applications maintaining permissions when the permissions were replaced. | Applications with fewer permissions could access components of a more privileged application. |
| Normal-level permissions are not shown to the user. | Permissions can be granted unknowingly. |
| Permissions with signature-based levels are not shown to user. | Permissions can be granted unknowingly. |

# Chapter 3

# Methodology

## 3.1 Application

The specific application to demonstrate cloud-smartphone interaction involved ray tracing a scene and performing image retrieval with the output of the ray tracer. This application was chosen because of the large amount of processing that was necessary and the size of results, which the user would likely wish to save to his or her device. Because of these details, the application benefitted from cloud support, and it required use of several Android and cloud security mechanisms. The specific application is described below.

1. Ray tracing: ray trace a scene. In this case, the classic Turner Whitted scene described in [23] was chosen. Advanced features of the ray tracer included rendering images with variable pixel resolutions, performing rotated-grid supersampling, and rendering a variable number of objects in the scene.

2. Library compilation: retrieve images from a local cache. This step was done to simplify the implementation, as similar images could be injected into the image library so a match would always be found. Images could have also been downloaded dynamically from the Web.

3. Image retrieval: compare the ray-traced scene to the images in the image library and find the closest match via an algorithm [24] provided by OpenCV [25].

Given the description, one could imagine its uses as a real-world solution, as a more flexible ray tracer would allow users to quickly describe an image or object (without an image of the actual object required) and find images that match from the Internet. The specific steps for image retrieval are described below. The actual steps performed are important because they affect performance.

1. Ray-traced (source) image was loaded as a format that was understood by the OpenCV library.

2. Images from the image library were loaded one after another. Images were converted to grayscale and were resized to the pixel resolution of the source image so that the output of [24] was a single number between -1 and 1, in which a value of 1 indicated

a perfect match. Then, [24] was used to compare the library images to the source image.

3. Image resources were freed after the comparison. If a new best image was found, the filename of the best image was saved to memory.

It should be noted that this method of image retrieval may not have been the most efficient. Retrieval could possibly have been completed more quickly and more accurately if, for each comparison, either the source or target image were resized to the pixel resolution of the smaller image. However, the images from the library were always resized to the pixel resolution of the ray-traced image. This allowed for more consistent statistics that showed the effects of pixel resolution even as the images in the library were changed. The code in the cloud software that performed image retrieval is shown in Appendix A.

## 3.2 Implementation

The cloud-smartphone system consisted of two distinct parts: a cloud application and an Android application. A visualization of the entire system architecture is shown in Figure 3.1. The Android application required two permissions: INTERNET and WRITE_EXTERNAL_STORAGE. The application consisted of three main components. A main activity was launched initially when the user opened the application. It included options for a username, a password, and the IP address of the server (described later). An asynchronous task was included that handled the connection to the cloud and ran in the background while the application waited for results from the cloud. Finally, a result activity was added that displayed the result returned from the cloud and it allowed the user to save it to external storage. Screenshots of the application as it transitioned through each activity are shown in Figure 3.2.

After the user entered the required information and requested a connection, the application sent a request to the cloud for a description of available cloud instances. If an instance with the expected cloud instance ID and the supplied IP address was found to be running, the application attempted to connect to it. These steps helped to ensure that connections were made only with valid servers. The code for the main loop of the Android application, in which the application authenticated its user with the server, sent its work request to the cloud, and received an intermediate result, is shown in Appendix D.

Intent filters were defined for each component in the Android application. The main activity was defined such that it could be launched only from the Android Launcher. The result activity's Intent filter allowed it to be launched only with the VIEW action and the DEFAULT category. Additionally, a data file had to be attached, which was required to have the MIME type *image/png*. If the data file was not available or if it could not be opened as an Android Bitmap object, then the activity did not attempt to open any image. The asynchronous task executed within the main activity's process; thus it did not require an Intent filter. Also, an Intent filter was not necessary for the result activity. If one was not defined, then the component would not be displayed as an option for generic Intent

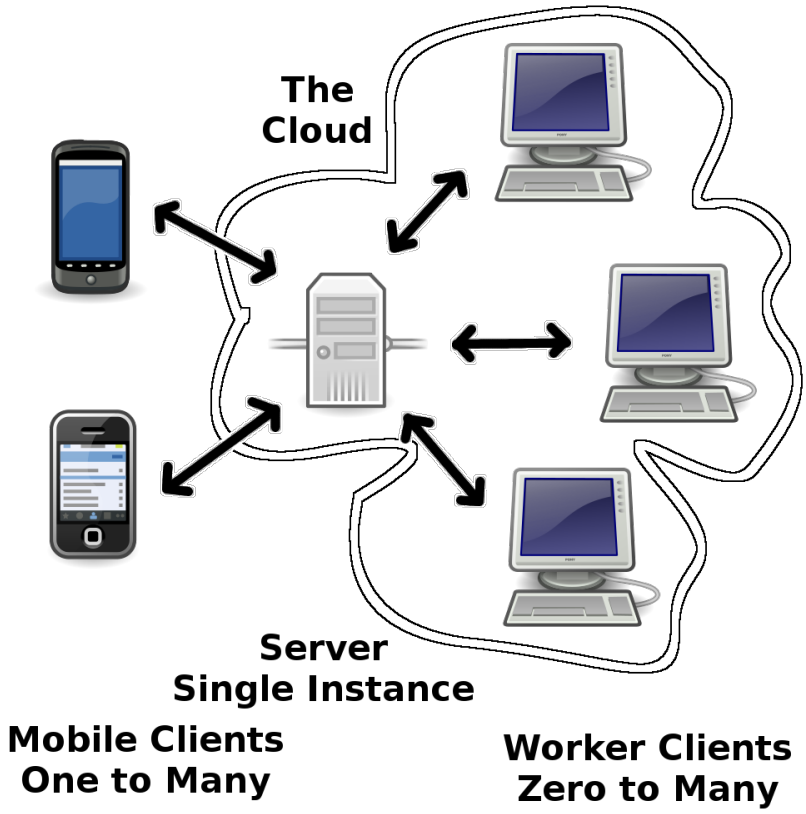


Figure 3.1: CloudSmart system architecture

requests. However, with the inclusion of the Intent filter, the component was available for other applications to launch to display images.

The cloud application consisted of two distinct parts: a client (also referred to as a worker and a worker client in this document), which performed work offloaded to the cloud, and a server, which managed connections to the cloud and served as an intermediate point for communication between the end user and worker clients. The server, which was limited to a single instance in the system, accepted (or rejected) new users and clients, removed disconnected users and clients, and forwarded communication from users to each client.

The server contained two subcomponents to perform these actions. The server launched a client handler for each connected worker client. Each of these handlers listened for communication from a single client and sent data to that client. Each handler executed within its own thread. Similarly, the server launched user handlers for each end user. Each of these handlers listened for communication from a single end user and sent data to that user. Like the client handlers, each user handler executed within its own thread. If no worker clients were connected, the user handler would perform any work offloaded to the cloud.

On the other hand, each worker client would perform any work offloaded to the cloud by any user. Clients performed their work independently and did not cooperate with each other. However, each client could be given different data with which to work. In the example application, each client could start with a different set of images in its library. After the completion of its work, each worker sent its results to the server, where the server would select the best result, save it, and forward it to the end user. The results were saved in case their transmissions to end users failed. It was important that saved files were managed correctly. The files always remained private, and they were deleted when they were no longer needed. The complete source code for the worker client class is shown in Appendix B.

Worker clients did not communicate directly with the end user. The server acted as an intermediate point for all communication between workers and end users. This design decision isolated mobile users and worker clients, and it allowed messages to be filtered in a centralized location before prior to communication with worker clients.

Challenge-Handshake Authentication Protocol (CHAP) [26] was implemented for the server to authenticate users and clients when they initially connected to the server. CHAP was originally designed for use with Point-to-Point Protocol, and it was used in the same fashion for this project: to authenticate a user with a server in which the systems communicated directly. Two benefits of CHAP are that users' actual secrets are not sent to the server and that the protocol resists replay attacks as a unique challenge is sent for each handshake for use in the one-way hash. Users were identified by a unique username chosen during registration. Each user's secret was an SHA-1 hash of his or her password, a string of human-readable text. SHA-1, which stands for secure hash algorithm 1, was endorsed by the United States government. The algorithm produces a 160-bit output for any input. The list of registered users and clients was stored in a text file in a private directory on the server. A text file was used because of the small number of expected users and clients. A database could have easily been used in its place. For additional security, the list or database of users should be encrypted in case the server was compromised.

Communication between clients, users, and the server was encrypted with 256-bit AES in CBC mode. AES stands for advanced encryption standard, and, like SHA-1, it is endorsed by the US goverment. The largest key size supported by AES, 256 bits, was used. CBC, which stands for cipher-block chaining, adds additional complexity because the output for each block depends upon the previous block. OpenSSL [27] was used for the C++ cloud server and client. Android's implementation of Java Cryptography Extension (JCE) [28] was used in the Android application. Keys were created by the algorithm described in [29] with an eight-byte salt and the concatenation of each user or client's username and hashed password as the key data. This selection of the key data ensured that keys used for each user would be unqiue. The key data was hashed five times with SHA-1 before its encryption with 256-bit CBC AES to obtain the key and initialization vector. The CHAP handshake used a predefined text string as the key data, but all other communication used the username and password hash string for the key data in order to obtain a unique key for each user, which prevented one user from deciphering another user's data. It should be noted that this method is susceptible to replay attacks; a remedy is explained in the next section.

An example execution of the application begins with a user's entering the required information on the Android interface. The application uses the information to authenticate itself with the cloud server. Upon successful authentication, the request is sent to the server at the specified IP address. If successful, the server creates a user handler. The Android application would then automatically send a work request to be received by the user handler. If no worker clients are connected, the user handler performs the work for the end user and immediately returns the result to the end user. Otherwise, the user handler forwards the request to any connected worker clients via their client handlers. Each client ray traces the scene and compares its ray-traced image to its library of images. Each comparison yields a number, referred to as a score, which corresponds to the relation between the ray-traced and library images. When the image retrieval completes, each worker sends its best score to the server, which passes the result to the corresponding user handler.

After the reception of each result, the user handler determines if it received enough results to choose the best one. If enough results have been received, the best one is selected and its source is noted. Finally, the user handler asks that source for its final result (the best-matching image, in the example application) and it receives the final result. The server could operate under two modes. If one more than half of the connected worker clients returned the same score, the user handler then requests the final result from one of the clients with that score. If no majority decision was reached, the user handler waits for all results to arrive and simply chooses the best score (this alone is the second mode). This code, implemented for the Android application, is shown in Appendix C. To get the best result, the user handler creates a request and passes it to the server component, which sends the request to the intended client. The client sends its result back to the server, which passes the result to the user handler, which saves the file to a private directory. This program flow as seen from the Android interface is shown in Figure 3.2.

For further protection, the file should be encrypted. If a worker client disconnects after it was asked for its best result, the client is ignored, the best score is determined again, and the next best client is asked for its result. This process is repeated until a result is returned or

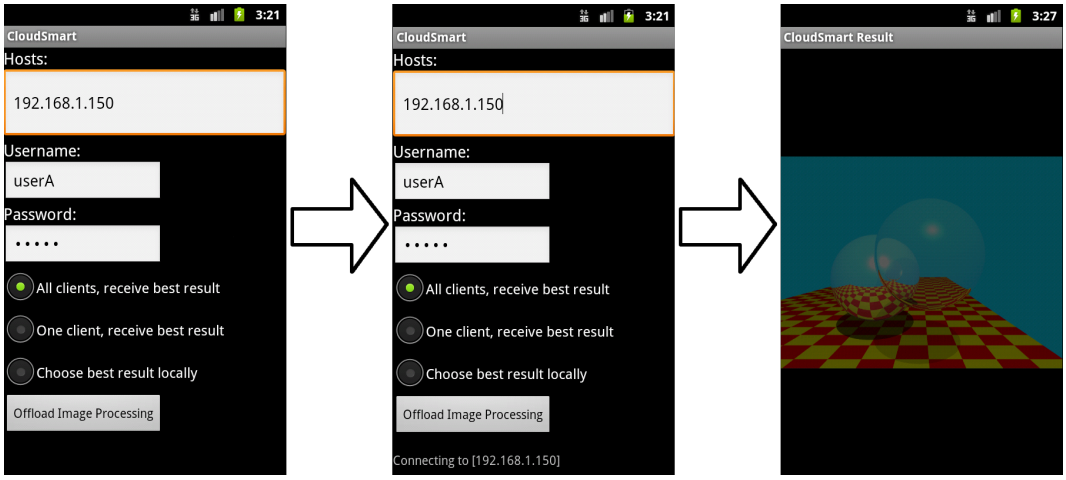


Figure 3.2: Screenshots of the Android interface as the program transitioned through three phases of execution. Th application began in an idle state waiting for information on the cloud from the user (left). After the user pressed the button, the work request was delivered, and the interface was updated accordingly (center). The application stayed in this state until the final result was received, at which point it was displayed (right).

until no clients are connected, at which point the user handler performs the work itself. The specific implementation of this feature was not perfect, and, under certain circumstances, the user handler would wait forever for a disconnected client to return a result. To combat this possibility, a reliable, low-cost watchdog as proposed in [30] and [31] could have been used to update client statuses and ensure result delivery.

As worker clients handled work for multiple users, it was necessary to ensure that users could not retrieve other users' results. Thus, each result was saved along with the ID of the user that requested it. IDs were managed only on the server. Additionally, clients communicated only with the server and did not accept any other incoming traffic. These two items combined with the authentication protocol ensured that no user could masquerade as another user to steal his or her results.

When the user received his or her result on his or her mobile device, the result was saved to the application's private storage folder. This protected the file from access by other applications that otherwise could upload the file to a server. An *export* option was available in the result activity that allowed a user to save results to external storage, where the files would be public and available to all applications on the device. Under normal circumstances, the application's private directory remains inaccessible even to the owner of the device. However, this is not the case on "rooted" devices. Thus, for additional security, files should be encrypted when saved to the private directory. However, it was not an option to save an encrypted file: if the application were uninstalled or the file were removed from the external storage, then the file would become unusable.

An additional configuration of the cloud software was authored in which the system was restructured to contain multiple servers on the cloud to which the mobile user connected.

Each server performed any work offloaded to it. In this scenario, no cloud instance chose a best result; rather, the mobile user received each intermediate result. In the application presented in the paper, these were the scores that corresponded to an image's similarity to the ray-traced image. From these intermediate results, the application on the mobile user's device could choose the best result. The advantage with this configuration was that the redundancy allowed the application to ignore artificial results. This version also circumvented the server in its usual role, which was a single point of failure if compomised.

## 3.3 Security features

With the Android application and cloud software developed, the system needed to be secured. This process required a selection of vulnerabilities to address. The most obvious choice of vulnerabilities to address included those that could be solved merely with design choices. This selection included basic methods to secure private data and to ensure expected operation. For instance, when saving a file to a directory, a handle to that directory must be opened. The design choice is whether it is better to save to a public directory or to a private directory. The next step in selecting security features was to select features that needed to be implemented, or those features that required more effort than simple design decisions. It was decided that these features should impact the same vulnerabilities addressed in the previous step. Of course, there were countless options for this stage of action, but several options could be considered obvious, such as the need for encryption and authentication. No effort was placed on patching exploits; even as one exploit is patched, many more exploits will be discovered, and the technologies in use will become obsolete. Rather, effort was placed on defining a secure foundation upon which future work can build. Certain vulnerabilities described in related work influenced the selection of vulnerabilties addressed in this work. These issues are summarized in Table 3.1.

A number of best practices were determined from this project's implementation. Certain practices are specific to Android. All private information including personal files and account data should be saved in private directories to protect it from rogue applications. Android provides Context.openFileOutput(String, int) [32], which opens a handle to a private directory. As previously detailed, files saved in private directories may not remain private if a device is "rooted." Thus, private files should be encrypted when saved, including any credentials, such as passwords and cloud credentials. For encryption, AES has been proven to be resilient against attacks [33]. Data sent over the air should be encrypted as well, as encryption is not performed automatically. Along with AES, RSA is a suitable candidate for these applications [34]. As part of CloudSmart, files were not encrypted when saved, but all communication was encrypted.

Android activities and services can return results; however, results may not be computed properly, and applications can crash if they use such results. Thus, applications should always check the *resultCode* value returned by an activity [35]. Similarly, return values of function calls should be checked, like the values returned by Intent.getData() [36], Intent.getStringExtra(String) [37], and BitmapFactory.decodeFile(String) [38]. No activities were used to return results, but all function return values were checked in CloudSmart.

Table 3.1: Known vulnerabilities addressed in this work

| *Specific Vulnerability* | *Actions Taken to Address* |
|---|---|
| Web browsers lack support for XML Signature and XML Encryption, allowing credentials can be moved from one machine to another to appear as a different user. | Save files to private storage [32]. |
| Cloud Malware Injection Attack | Use multiple instance to provide redundancy [40] and avoid single points of failure. |
| Developers cannot enforce that called applications hold or do not hold certain permission. Similarly, developers cannot ensure that launched applications will not handle supplied data maliciously [46] [11]. | Remain open issues. Can mitigate by calling applications my name only and refraining from passing sensitive data betwen aplications. |
| Applications that hold dangerous permissions can create backdoors for unprivileged applications [12]. | Remains an open issue. |

Regarding interactions between applications, permissions [19] should be required where necessary. These are used to notify users of applications that may perform dangerous actions. No permissions were required for this project as no public components performed dangerous actions. Also, meaningful Intent filters [20] should be defined for all public components. Intent filters limit the ways in which activities and services can be launched. The Intent filters created in this project were described in the pervious section.

Other practices relate to overall systems. Secure authentication methods should be used for systems in which multiple users must be remembered by the system. Such systems should be resistant to many different attacks and help to secure users' privacy. The project described here used the protocol described in [26]. TLS [39] should be used to authenticate any servers, which allows users to be confident that they connect to the correct servers. TLS was not implemented for this project as the expected complexity of implementation was thought to be quite high. Finally, if instances are compromised or malicious instances are inserted into a system, they can send invalid results to users. Thus, multiple clients should be used to provide redundancy to confirm results as in [40]. For the same purposes and also for reliability, single points of failure should be avoided. Redundancy was accomplished in CloudSmart with the additional feature of the software that allowed the Android application to select the best result from multiple servers.

A number of security features were proposed for this thesis. It is very important that private information is not recorded with logging functions. In general, before an application is released to the market, all logging should be disabled [41]. However, it is *very* important that all calls to logging functions containing private information such as user accounts, names, and passwords be removed as any application with the READ_LOGS permission

can execute Android's LogCat program like a shell script with the RunTime class [42]. In this project, all logging was disabled in versions distributed to other users.

Additionally, it is proposed that data should initially be saved to private directories with the option provided to users to export their data to public directories, such as external storage. It is wise to add this option as users may wish for personal data, such as audio recordings and pictures, to remain private to the application that created it. The extra option is especially beneficial if rogue applications are present on users' devices. Also, as the user is unaware of most functions by default, users should be notified of any important events, such as saving data to public directories and uploading or downloading files to or from the Internet. Android provides the Toast class [43], which is an option for quick notifications that was used in CloudSmart.

Finally, cloud commands such as *euca-describe-instances* [44] and *ec2-describe-instances* [45] can be used to verify instances on which servers reside. These functions are beneficial because every cloud instance on which servers reside has an image ID and an instance ID. The instance ID for each server in a system may change, but its image ID will not change. Thus, if a server's image ID (or possibly its instance ID, depending on the system) is different than expected, then the user should ignore the server. While these methods can be useful, TLS remains an Internet standard as it has been proven to be effective. In CloudSmart, the *euca-describe-instances* command was used to validate instances.

Larger security gaps exist as well. One such vulnerability is that developers and users cannot prevent other applications from uploading public files to remote servers. This is a concern especially on Android devices, in which private files, such as photographs and audio recordings, are very likely to be present and saved in predictable locations. A number of issues contribute to this vulnerability. It is not possible to ensure that launched applications will not handle supplied data maliciously, such as saving data to public locations or calling other applications to handle data. Similarly, developers cannot enforce that called applications hold or do not hold certain permissions [46]. For instance, an application that passes its data to an unspecified third-party application cannot ensure that the application does not hold the INTERNET permission, which would allow the application to upload the data anywhere. This vulnerability was addressed with Saint [11], but its solution requires modifications to Android that have not yet been implemented. One suggestion is that only trusted applications should be called. Finally, files saved in public locations can be silently uploaded by any application with the INTERNET permission. There is currently no method to avoid this vulnerability short of saving data to private directories, but private directories prevent users from accessing the data without the application that saved it. The previous suggestion to save data to private directories initially while allowing users to export their data to external storage may be helpful here. If files are saved to public directories by default, it is recommended that obvious naming schemes for such files are avoided.

An interesting vulnerability that is difficult to solve is also related to Android: opening files from remote clients can be potentially dangerous. Even if files are not dangerous to Android devices, they can be dangerous if sent to PC users. One possible solution is to scan files for viruses. However, no popular antivirus application's public documentation suggests it can scan specific files on-demand. Even if an application supported that functionality, the antivirus application could not access private files to scan them under

Android's security model. Thus, the only real solution is to build antivirus features into an application. However, this adds to the size of applications, and it increases the overhead for developers as they would need to integrate antivirus frameworks into their applications. An interesting solution is to avoid local antivirus checks and offload this work to the cloud.

Applications that hold dangerous permissions can create backdoors for unprivileged applications. This vulnerability undermines the permissions system in that even if an application requires that calling applications hold certain permissions, an application that holds those permissions can create activities and services that can be called by other applications with no permissions. This vulnerability was addressed by Kirin [12], but that project required modifications to Android that have not been implemented. A solution that can be employed by an developer for his or her applications has yet to be introduced.

Data received over the air can crash applications if they are invalid. As previously mentioned, return values of functions should be checked. In addition, Android offers features that can be used to validate input, such as the InputFilter interface [47] and the *inputType* attribute [48]. However, these options are for user-interface fields only. Data validation frameworks exist for Java and C, but no such framework is targeted specifically to Android. The vulnerabilities and recommended practices described in this section are summarized in Tables 3.2 and 3.3.

Table 3.2: Summary of vulnerabilities encountered through development of this project.

| *Vulnerability* | *Solutions* | *Open Issues* |
|---|---|---|
| Any public file can be silently uploaded to remote servers by any application with the INTERNET permission. | Save files to private storage [32]. | No solution for public files. |
| Cannot ensure that launched applications will not handle supplied data maliciously. | Saint [11], but it is not implemented in Android. Can mitigate by calling applications by name only and refraining from passing sensitive data betwen aplications. | No current method to limit called application privileges. |
| Developers cannot enforce that called applications hold or do not hold certain permissions [46] [11]. | Saint [11], but it is not implemented in Android. Can mitigate by calling applications by name only and refraining from passing sensitive data betwen aplications. | No method to limit called application privileges. |
| Opening files from remote clients can be dangerous. | Build antivirus functionality into an application. | Building antivirus functionality into each application creates unnecessary complexity. |
| Applications that hold dangerous permissions can create backdoors for unprivileged applications. | Kirin [12], but it is not implemented in Android. | No current method to prevent such backdoors. |
| Invalid data can crash applications. | *ResultCode* should always be checked for activities that return results [35]. Function return values as with [36], [37], and [38] should always be checked. Android provides IntentFilter [47] and *inputType*. | IntentFilter and *inputType* [48] are useful for UI fields only. No data validation frameworks are targeted specifically to Android. |
| Files saved in private directories may not remain private if a device is “rooted.” | Encryption such as AES can be used [33]. | None. |

Table 3.3: Summary of vulnerabilities encountered through development of this project (continued).

| *Vulnerability* | *Solutions* | *Open Issues* |
|---|---|---|
| Access to dangerous features needs to be protected. | Android permissions system [19] should be used. | Applications that hold permissions can create backdoors (see above). |
| Access to public application components needs to be limited. | Android provides Intent filters [20]. | None. |
| There is no automatic notification of important actions such as uploading files or saving files to public locations. | Toasts [43] are convenient and simple. | None. |
| Personal information can be read in system logs by other applications and users. | Disable logging prior to application release [41]. | None. |
| Data in systems with multiple users needs to be kept private. | Use an authentication system that is resistant to various types of attacks such as CHAP [26]. | None. |
| Server identities need to be verified. | TLS can be used to verify servers [39]. Cloud commands like [44] and [45] can be helpful. | Vulnerabilities are present within TLS. |
| Invalid cloud instances can poison results. | Use multiple instance to provide redundancy [40]. Avoid single points of failure. | None. |

# Chapter 4

# Evaluation

## 4.1 Experiments

Performance of the developed software was evaluated with regard to execution time, battery usage, and memory usage. An Android application was developed as a foil to the cloud-supported software, and its performance was evaluated in the same areas as the cloud-supported application in order to provide a baseline for performance. It performed the ray tracing and image retrieval tasks using the same algorithms that the cloud-supported application performed, but with only the mobile device for processing. The application's library of images for image retrieval was located on the smartphone's external storage.

To measure execution time and memory usage, at least ten trial executions of the applications were completed. Execution time was measured as the delay between the transmission of the initial request to offload work and the reception of the final result by using Android's SystemClock.elapsedRealtime() function.

Android offers many methods by which memory usage can be measured. Multiple methods are available via the LogCat function of the Android Debug Bridge. Android also maintains a file available in the *proc* filesystem. Programmatically, a MemoryInfo object can be obtained by an application by calling the ActivityManager.getProcessMemoryInfo(int[]) function, which allows the application to survey its memory usage. The MemoryInfo object contains three metrics: shared dirty pages, private dirty pages, and proportional set size. Each metric was available for dalvik (Android's process virtual machine, which is simalar to a Java virtual machine), the native heap, and "everything else" (other) [49]. Shared dirty pages was ignored because of the indication that it accounted for memory shared between several applications. Thus, the private dirty pages and proportional set size metrics were recorded. For each metric, the dalvik, native, and other values were summed. Figure 4.1 shows average memory consumption for ten trials of the local-processing application for both separate parts of the execution. Clearly, there was a correlation between the metrics, and this trend was present on all subsequent trials. Because of this trend, only the private dirty pages metric was used to compare memory usage for all tests. Memory was measured throughout the applications' executions by obtaining MemoryInfo objects at various points during the executions.

A curiosity with memory usage is shown in Figure 4.2, which shows the full memory statistics for a single execution of the local-processing application. The level part of the

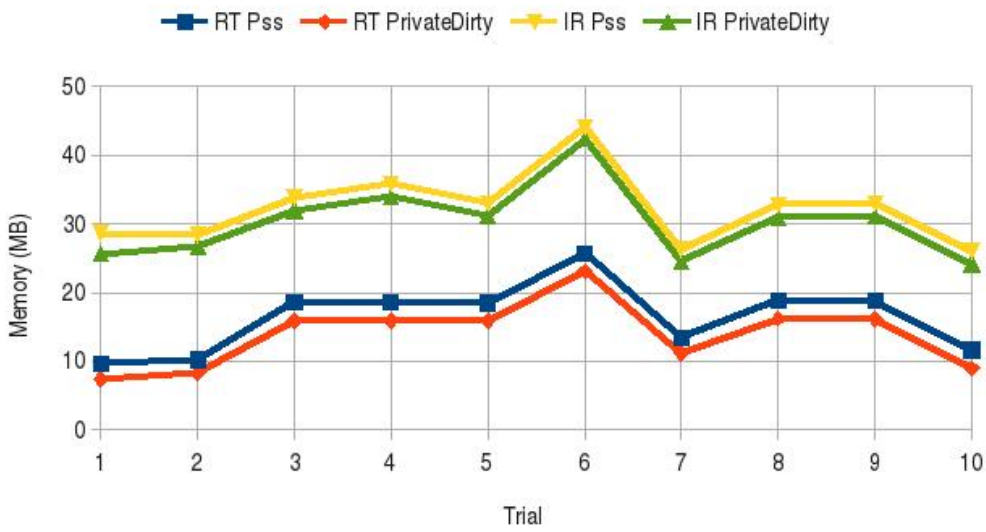


Figure 4.1: Average memory usage in the local-processing application over ten executions with 118 images and 800x600 ray-traced images. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

chart shows the memory usage for the ray tracing portion of the execution. As the image retrieval portion began, the memory usage began to rise to more than double the memory usage of the first part of the application. This behavior was found on all tests of the local-processing application. With larger images, the application crashed during the second phase. Although exhaustive attempts were made to manage memory better, but the trend remained.

Battery usage was measured through creating a BroadcastReceiver for Android's BatteryManager. It was discovered that battery level was reported with very coarse granularity: 0 to 100 % at 10 % increments. Because of this, it was very difficult to record accurate battery statistics. To combat the coarse granularity, enough trials of each application were run such that battery level decreased by 20 % of the full scale. This decrease was achieved in order to obtain a more accurate representation of battery usage, as it was assumed that when the battery dropped to the next lowest level, it was exactly at that level (*e.g.* if the battery level decreased from 90 % to 80 %, the battery life was assumed to be exactly 80 % of the full scale). Additionally, as many background services as possible, such as WiFi, GPS, and data syncing, were disabled while running the applications. Because percentage of battery life depends on the specific battery used, the percentages of battery life were multiplied by the Watt-hour (Wh) rating listed on the battery.

The issue with Android's coarse granularity for battery usage is exemplified in Figure 4.3, which shows the average reported battery level from several executions of the

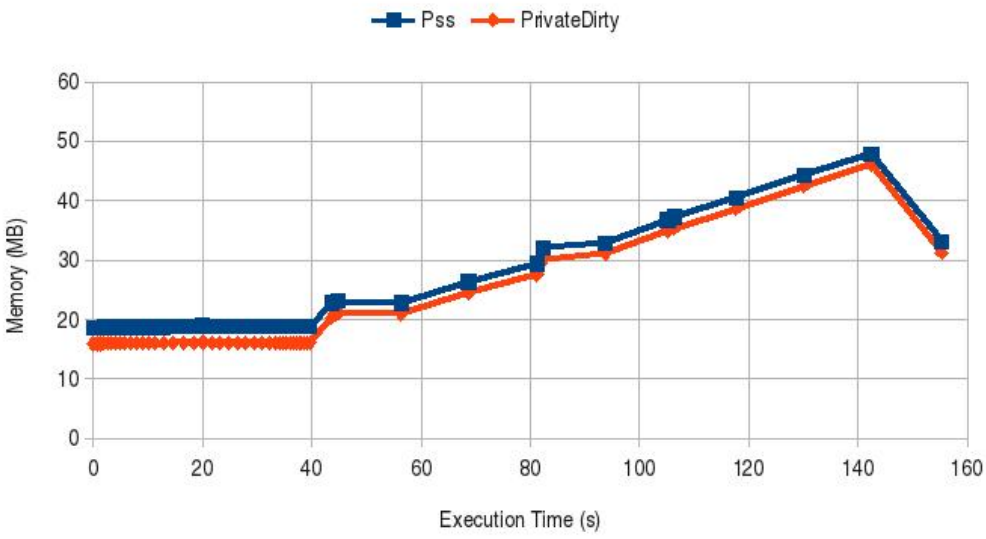


Figure 4.2: Memory statistics from one representative execution of the local-processing application with 118 images and 800x600 ray-traced images.

local-processing application. The chart illustrates the issues posed by the collection of battery information. For several executions of the application, the reported battery level did not change. Rather than decreasing smoothly over a number of trials, the coarse granularity in which battery level was measured indicated that battery level dropped 10 % only during specific executions.

Each performance metric was measured on both the cloud-supported and local-processing applications. Two varibles were present in these measurements: the number of images in library (that were compared to the ray-traced image) and the pixel resolution of the ray-traced image. Changes to the number of images did not affect the ray tracing portion of the execution, but differences in the ray-traced image's pixel resolution affected both parts of the application as the images in the library were resized to the dimensions of the ray-traced image. Finally, the measurements were also obtained from the cloud-supported application without the encryption and authentication features in place and from the software with the feature enabled in which results were selected locally.

## 4.2 Test setup

Results were obtained on a Motorola Droid X on which Motorola's official Android 2.3.3 firmware [50] was installed. The approximate capacity of the device's battery was 5.7 Wh. Amazon's Amazon Web Services Android SDK [51] was used to provide cloud commands for the Android application. The cloud that was used ran Eucalyptus 2.0.3 [52] on CentOS

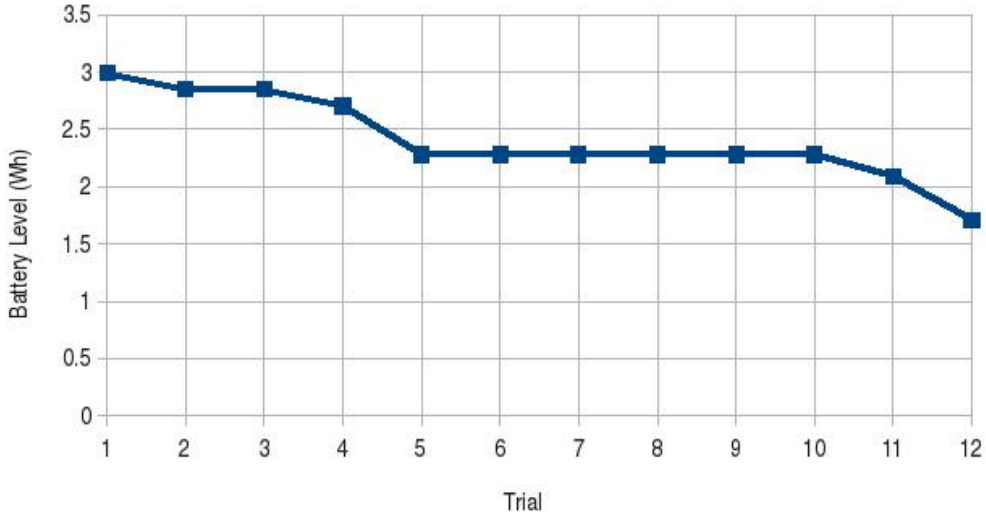


Figure 4.3: Average reported battery level over several consecutive executions of the local-processing application with 118 images and 800x600 ray-traced images.

5.6 and included three Intel quad-core machines, one of which (the front-end) hosted the cloud controller, Walrus, cluster controller, and storage controller. The other two machines were node controllers. Because of limitations imposed by the campus network, the front-end machine was connected to a router in order to allow IP addresses on the same subnet as the front-end machine to be assigned via DHCP to end users. End users were required to connect to a wireless network hosted by the router. The node controllers and front-end were connected via a private switch that was not connected to the Internet. This configuration, shown in Figure 4.4, supported Eucalyptus's *managed* networking mode, which was used because it offered the highest degree of virtual-machine isolation. However, any networking mode could have been used.

For image processing, OpenCV 1.1 was used in the cloud-supported application. This older version because the cloud images' host operating system was CentOS 5.3, which supported only up to OpenCV 1.1. The local-processing application used OpenCV 2.3 for image processing [25].

Measurements were taken only on the Android devices because the goal was to analyze performance on the smartphones and the effects of offloading work to the cloud. No statistics were recorded on the cloud.

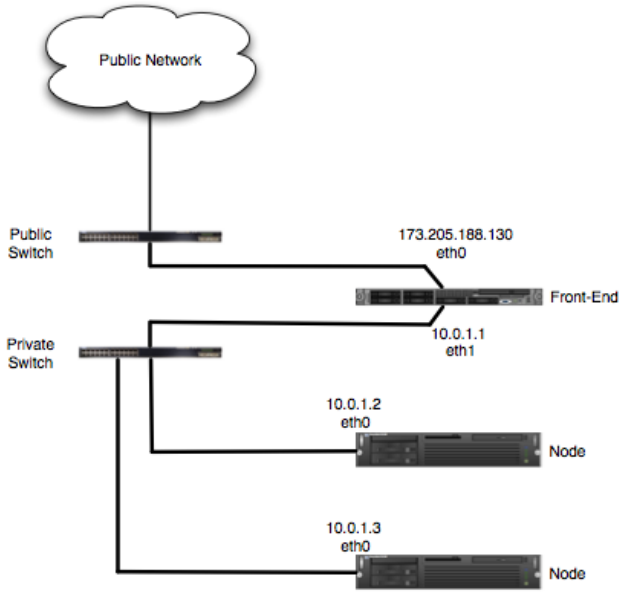


Figure 4.4: Network architecture of the cloud [4]. The listed IP addresses are example addresses and not the actual addresses used in the system.

## 4.3 Results

It was determined that current Android smartphones cannot support resource-intensive applications such as the one studied in this thesis. While there is no time limit to the run-time of an application, there are certainly limitations to how long users will wait for results. On average, the local-processing application's execution time was much longer than that of the cloud-supported application. The average execution time for the tests on the cloud-supported application did not exceed 22 seconds. On the other hand, the local-processing application consumed 7 minutes on average for the heaviest workload issued during testing.

While execution time increased roughly linearly with respect to pixel resolution of the ray-traced image and with respect to the number of images to which the ray-traced image was compared, it was found that execution time of the local-processing application increases much more quickly than the cloud-supported application as the workload grows. As the number of images in the comparison library was varied, the slope of the local-processing application's execution time was 1.04, while that of the cloud-supported application was 0.06. As the pixel resolution of the ray-traced image was varied, the slope of the local-processing application's execution time was $3.92 \times 10^{-4}$, while that of the cloud-supported application was $1.89 \times 10^{-5}$. These results are shown in Figures 4.5 and 4.6.

Current Android smartphones require too much battery power to support applications like the one described in this paper. Each execution of the local-processing application with 1200-px by 900-px images used on average 0.143 Watt-hours (Wh) per execution (single ray tracing and image retrieval). At this rate, the application could be run 40 times from

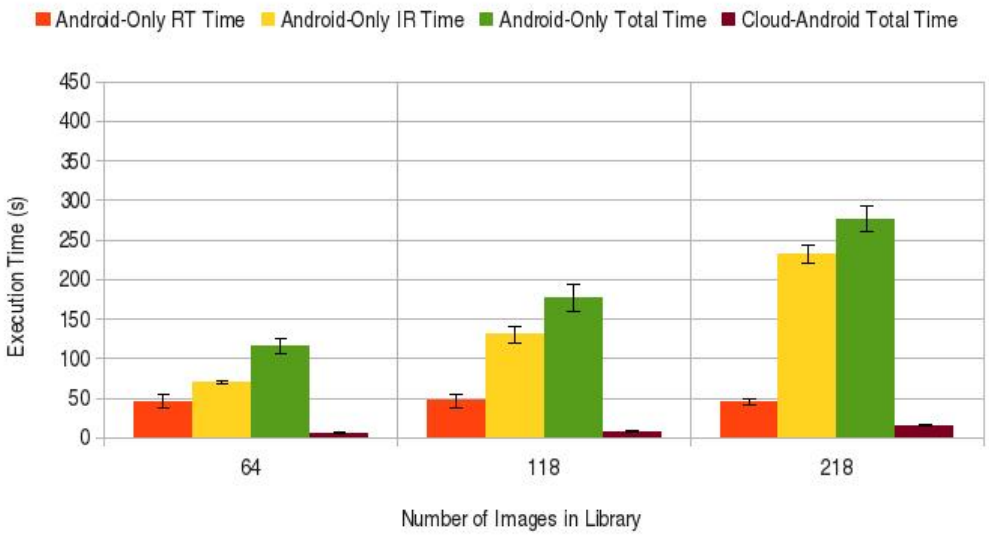


Figure 4.5: Average execution time in the local-processing application and the cloud-supported application over ten or more executions with 800x600 ray-traced images and a varying number of library images. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

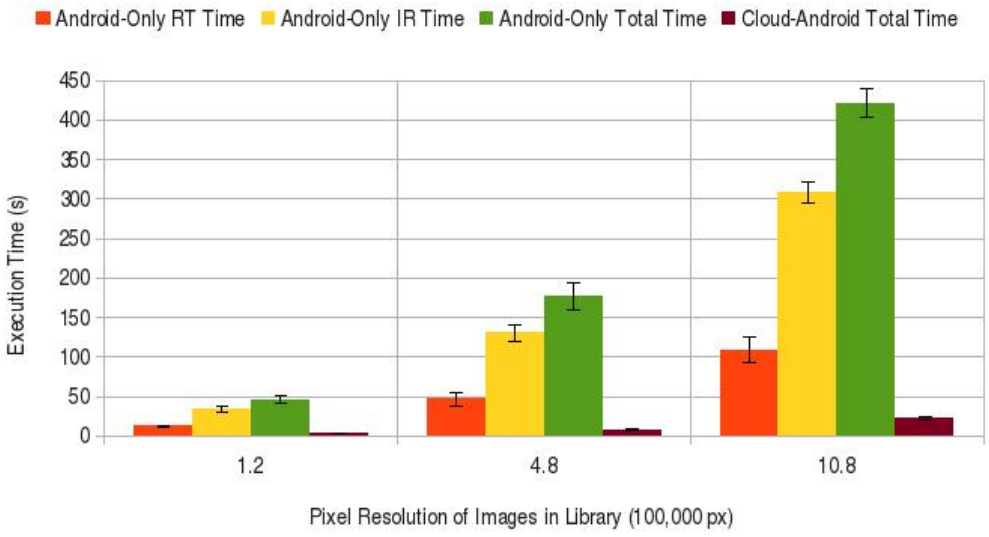


Figure 4.6: Average execution time in the local-processing application and the cloud-supported application over ten or more executions with 118 library images and a varying pixel resolution. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

a full charge until the entire battery power was exhausted. Most users would not want to run the application 40 times over a single charge, but even two to three times could reduce the battery life by a half hour. On the other hand, the cloud-supported application required at most 0.010 Wh, or 7 % of the highest battery usage of the local-processing application. Figures 4.7 and 4.8 show the battery usage results.

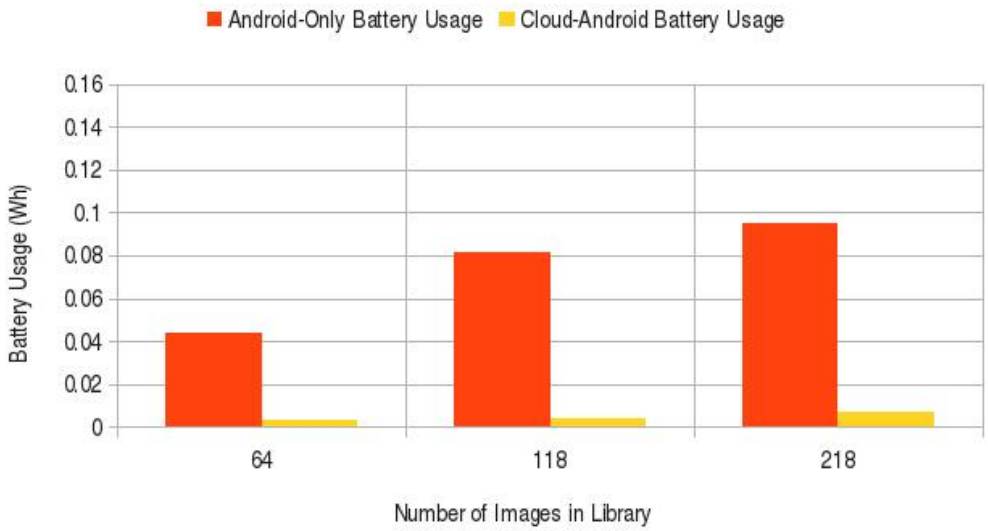


Figure 4.7: Average battery usage per execution in the local-processing application and the cloud-supported application over ten or more executions with 800x600 ray-traced images and a varying number of library images. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

Finally, current Android smartphones cannot support the memory (RAM) requirements of resource-intensive applications like the one described in this thesis. Frequently, the local-processing application crashed while ray tracing large images or while performing image retrieval on large images (1600 px by 1200 px and larger). The issues with crashes during the image retrieval phase were more severe and noticeable on images that even the ray-tracing portion could handle. The issues with memory consumption of the image retrieval phase might be addressed in future versions of the OpenCV image library. On the other hand, the limitations of the ray-tracing phase can be overcome only by increasing the memory that can be reserved by applications.

Average memory consumption did not increase as dramatically as execution time when workloads were increased. Interestingly, higher memory consumption was reported when the comparison library contained fewer images than when the amount was increased. This anomaly could be due to garbage collection, which perhaps did not run as aggressively as when the memory requirements were more intense. Still, the average memory consumption for the cloud-supported application was much lower than the local-processing application;

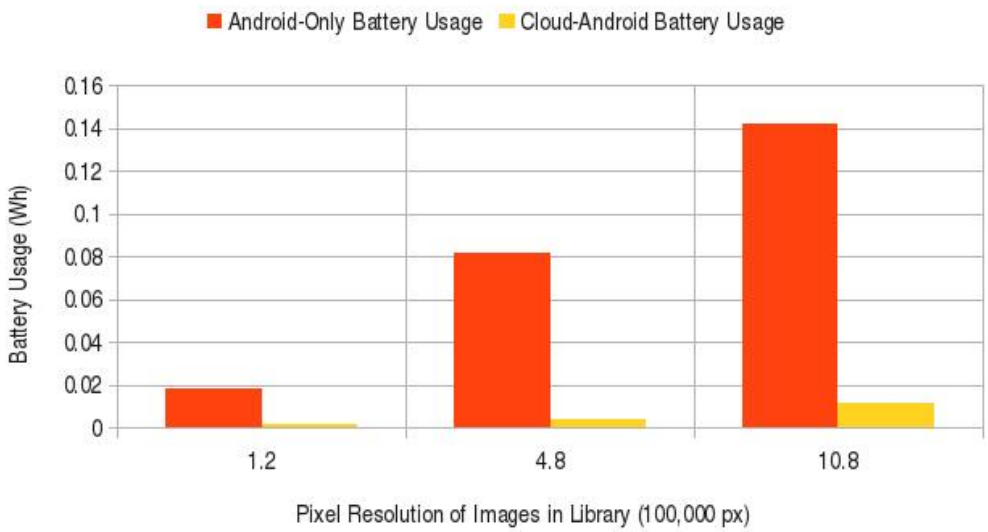


Figure 4.8: Average battery usage per execution in the local-processing application and the cloud-supported application over ten or more executions with 118 library images and a varying pixel resolutions. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

it required only between 6 and slightly over 7 MB of memory. On the other hand, average memory consumption for the image retrieval portion of the local-processing application reached 40 MB, and it was frequently between 30 MB and 35 MB. Average memory usage results are shown in Figures 4.9 and 4.10.

While the pixel resolution of the ray-traced image was varied in the tests performed, the complexity of the ray-traced image was not changed in any test as the contents of the ray-traced scene remained unchanged. Image complexity was not included as a variable because it was not expected to affect results. Figure 4.11 shows a comparison of average execution time of the cloud-supported application with varied image complexity. As shown, the difference between the ray-tracing and total execution times changed minimally as the number of objects in the rendered scene was varied from 3 to 50 objects. The time required for ray tracing was obtained by executing the ray tracer independently of the cloud-supported application. Another caveat is that the object placement was static for the first three objects placed in the scene; additional objects added to the scene were placed at random locations. Also, as the image complexity was increased, the template matching algorithm had a tendency to suggest that the smaller 7-KB to 15-KB images from the Caltech 101 library were the best matches, rather than the extra ray-traced images, which were approximately 70 KB in size. These details could have contributed to the small discrepancies in the differences between execution times.

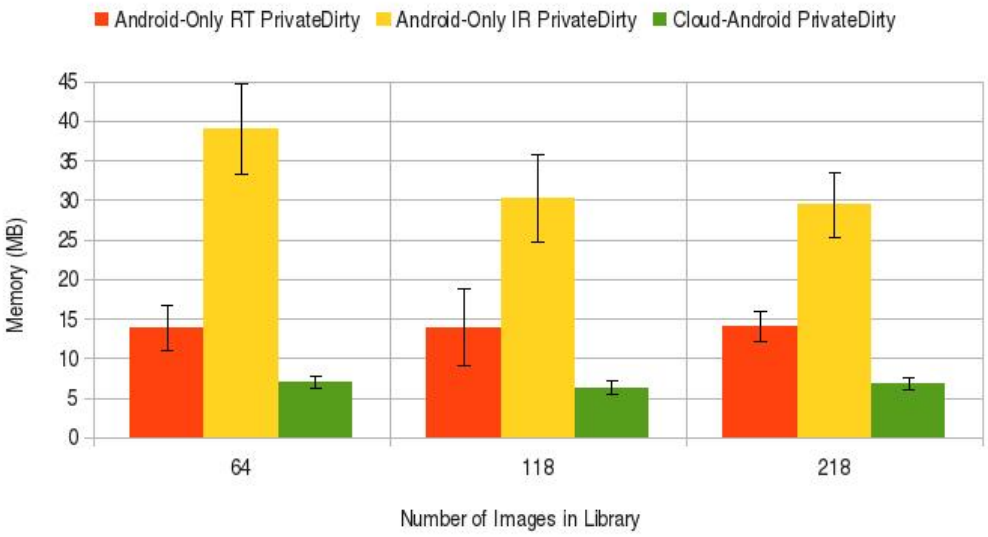


Figure 4.9: Average memory usage in the local-processing application and the cloud-supported application over ten or more executions with 800x600 ray-traced images and a varying number of library images. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

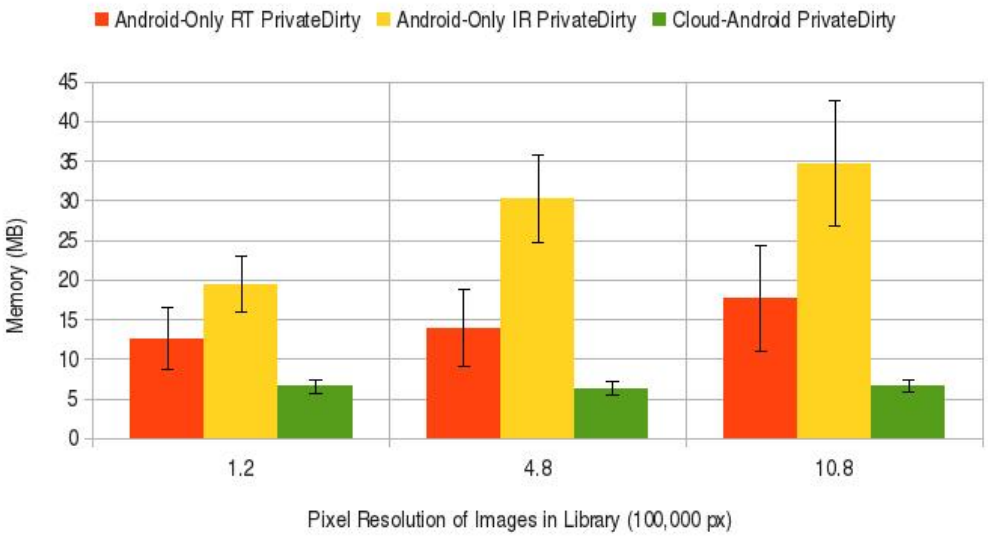


Figure 4.10: Average memory usage in the local-processing application and the cloud-supported application over ten or more executions with 118 library images and a varying pixel resolutions. RT indicates the ray-tracing portion of execution while IR indicates image retrieval.

To summarize the results presented thus far, it is fair to state that image complexity has little impact on the three metrics studied. It does affect ray tracing, however. Pixel resolution can affect both phases of execution, while the images within the comparison library have no effect on ray tracing. In the tests performed, battery usage and memory usage were affected most heavily by the pixel resolutions of images. On the other hand, execution time appeared to be most strongly affected by the number of images in the comparison library. However, the highest number of images included in the comparison library was 218 images in the tests performed. In a real-world application, the user would likely want the source image compared to thousands or millions of images, rather than to a few very large images. Thus, it would be very important to limit pixel resolution to the smallest resolution that returned the best results most efficiently.

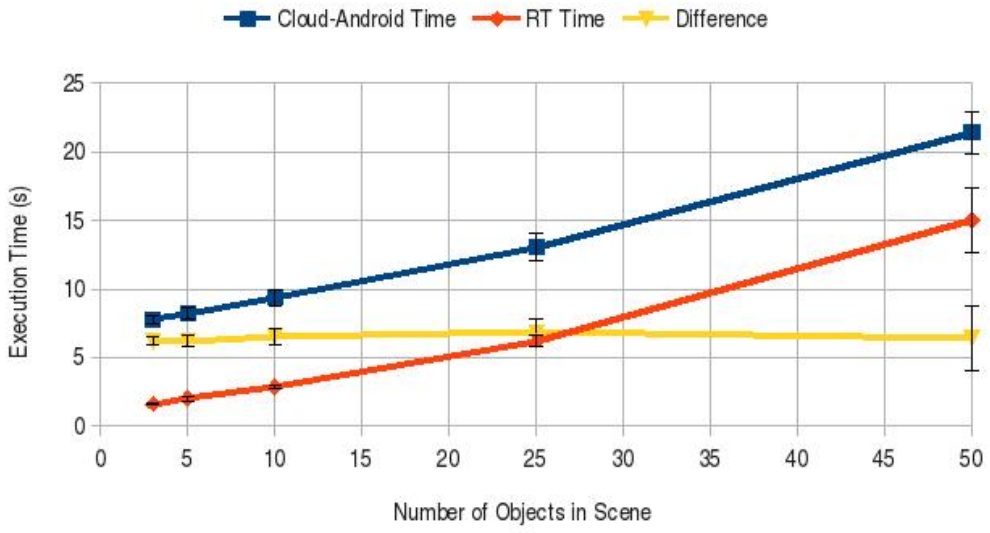


Figure 4.11: Average execution time in the local-processing application with 118 library images and 800x600 ray-traced images compared to the average time required for ray tracing (denoted by "RT") and the difference between the two times. The number of objects placed in the scene was varied.

The impacts of additional security features were measured together and were determined to be negligible. Android-specific features such as saving files to private locations, requiring permissions, and defining Intent Filters impose no noticeable overhead. Thus, the cloud-supported application was tested both with encryption and the authentication system and without those features. The difference in average memory consumption was less than 1 MB. According to the execution time statistics, the application with authentication and encryption performed over one second faster. However, both times are within range of each other, accounting for the standard deviations. Table 4.1 summarizes the results.

Table 4.1: Comparison of cloud-supported applications by security features (118 800-px by 600-px images)

| | *With Encryption and Authentication* | *Without Encryption and Authentication* | *Best Result Chosen Locally* |
|---|---|---|---|
| *Average Execution Time (s)* | 7.78300±0.30332 | 8.82592±0.96071 | 8.82517±0.49934 |
| *Average Battery Usage (Wh)* | 0.00413 | 0.00408 | 0.00424 |
| *Average Memory Usage (MB)* | 6.29185±0.81932 | 5.84003±1.21431 | 6.11826±0.94881 |

The additional configuration in which the application on the mobile device received intermediate results and selected the best result was also tested. This testing was performed with three servers on the cloud to which work was offloaded. Intermediate results were required only from one more than half of the servers (two, in this case) if the results were the same. Otherwise, the user's application waited for the results from all of the servers and chose the best result. With regard to execution time, this configuration of the application performed slightly slower than the other configuration. This is a logical result as sending intermediate results to users and receiving users' requests for final results added an extra overhead. Battery usage and memory usage were on par with the other cases. Given these results, this configuration was perhaps the best configuration tested in this project, as the security of authentication and encryption were maintained in addition to the provided redundancy. However, this scenario required communication with additional servers. Table 4.1 summarizes these results as well.

The effects of background traffic were studied. In a real-world situation, traffic in addition to the communication for a single user would exist, perhaps in the forms of communication with other users and traffic from other applications on the same network. Traffic was modeled by a simple exchange of a string between a user and a worker client repeated over a periodic interval. The worker client would copy and reverse the string a number of times until the message it had to send back to the user was up to 1024 bytes. Only results for execution time are shown. Battery usage and memory usage statistics were similar for each scenario.

It was found that the cloud software could withstand traffic of up to slightly over 20 KB/s per worker client (that figure represented the data sent from the worker clients). Faster transmission rates resulted in crashes of the user clients that generated traffic as send and receive buffers overflowed. Such crashes indicates that the cloud software would need to be refactored in order to withstand excessive traffic if the software were released as a production system. However, it was still true that the existing system could handle a number of users simultaneously, as the chances of many users (on the given system with the network architecture used during testing) simultaneously requesting offloaded work was quite unlikely. Requests for work were short, only on the order of bytes for a single request; thus many requests could be saved to a buffer without interruption of the software. The results from which these conclusions were drawn are shown in Figure 4.12. The corresponding traffic transmission rates for each scenario are shown in Figure 4.13.

The results shown in Figure 4.12 confirm that background traffic descreases performance. However, the descrease was only slight, on the order of 10 to 20 ms. An interesting outcome was that the case of all clients performing work with background traffic required slightly less time than the case of any single available client performing work. On the other hand, these values are well within range of each other when their standard deviations are taken into account.

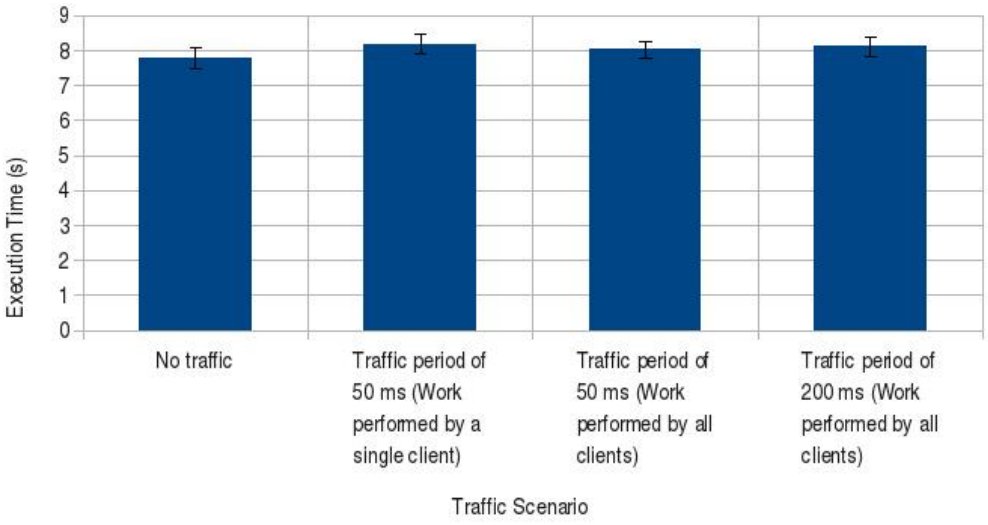


Figure 4.12: Average execution time in the local-processing application over ten or more executions with 800x600 ray-traced images and 118 library images. The amount of background traffic was varied.

One remaining question involves the effects of network latency on the system in a real-world application. For the testing performed, the front-end cloud node was behind a wireless router and the other nodes were connected to it via a gigabit switch. Meanwhile, the Android smartphone connected directly to the wireless network broadcast by the router. In a real-world setup, much more complicated routing would need to take place between the phone and the cloud.

It is believed that the communication between the cloud and the smartphone and its delay would be comparable to that of loading a Web page on a mobile device. Only two transmissions are made between the cloud and the smartphone. The initial authentication and work request messages are short (no more than a kilobyte). The remaining communication is the transmission of the final result, which in the cases tested is an image, such as the many images present on popular Web sites.

Network delay could also be an issue on the cloud side. It is possible that slower switches would be used or the route could be much more complicated. However, few cloud instances are required, and they could easily execute in the same datacenter. They could

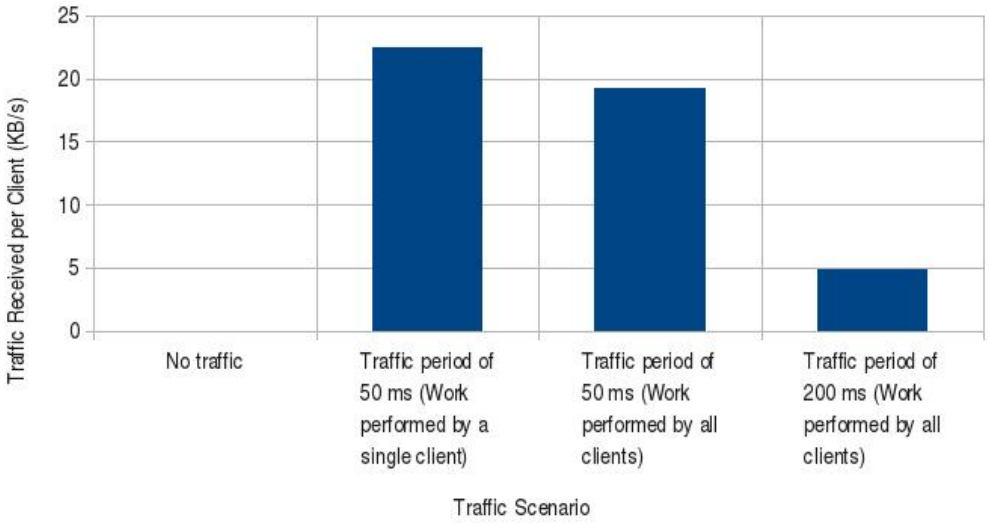


Figure 4.13: Average rates of transmission for data received by user applications that emulated background traffic. Each case corresponds to a result shown in Figure 4.12.

execute in virtual machines on the same physical machine, but this may be less likely as a goal of the cloud is to distribute load evenly. Additionally, there is not a great amount of communication between cloud nodes. A small amount of data (less than a kilobyte) is sent between nodes for the purposes of authentication, requests for work from clients, and transmission of initial results. Only one client returns its larger final result, an image in this case. Again, it is unlikely that the delay in the exchange of these data would exceed the delay present in the download of a Web page from a mobile device.

A summary of the effects on performance is divided into two sections: effects on the different phases of execution and effects on performance metrics. The effects on the ray tracing and image retrieval phases are recorded in Table 4.2. Pixel resolution of images affects both phases, while the number of images in the library affected image retrieval performance, and variations in image complexity affected only ray tracing. Effects on each performance metric are summarized in Table 4.3. It was diffcult to name a single variable as having the greatest effect on each metric, but analyzing the slopes of recorded data revealed trends. Execution time was most affected by the number of images in the image library, whereas pixel resolution of the ray-traced images and images in the library affected battery usage and memory usage most greatly. This conclusion is logical as higher resolution images require more memory, which leads to higher battery usage. However, if the example application were implemented as a real-world application, it would be very important to find the pixel resolution that achieved the best balance of efficiency and accuracy. It would also be important to maximize the number of images in the comparison library so that well matching images could be found in many cases.

Table 4.2: Comparison of effects on phases of execution (↑ denotes an impact, and ↔ denotes no impact)

| | *Number of Images in Library* | *Pixel Resolution of Images* | *Image Complexity* | *Background Traffic* |
|---|---|---|---|---|
| *Ray Tracing* | ↔ | ↑ | ↑ | ↔ |
| *Image Retrieval* | ↑ | ↑ | ↔ | ↔ |

Table 4.3: Comparison of effects of performance metrics (↑ denotes the most impact, ↗ denotes some impact, and ↔ denotes no impact)

| | *Number of Images in Library* | *Pixel Resolution of Images* | *Image Complexity* | *Background Traffic* |
|---|---|---|---|---|
| *Execution Time* | ↑ | ↗ | ↗ | ↗ |
| *Battery Usage* | ↗ | ↑ | ↗ | ↗ |
| *Memory Usage* | ↗ | ↑ | ↔ | ↔ |

# Chapter 5

# Conclusions

Cloud usage and smartphone usage continue to grow. Meanwhile, smartphone performance increases, and the devices become ever more connected, opening smartphones to a host of new applications once impossible on mobile devices. However, smartphones still suffer with regard to battery life. Thus, the incentives and ability to offload smartphone tasks to the cloud are greater than ever. However, many questions about the security of this marriage are unanswered while the number of malicious smartphone applications have climbed. The work presented in this thesis is among the first specifically to address security issues of interaction between smartphones and the cloud. This work focused on integrating vulnerabilities from a number of sources rather than discovering new issues. Most attention was placed on issues in the Android operating system.

The research presented is the first in this area to analyze the security of cloud-smartphone interaction. A simple but realistic system that offloaded a single task, ray tracing with image retrieval, to a cloud was designed and implemented. Future work will concentrate on extending the system to generalize it and allow it to offload a variety of commonly-performed smartphone functions.

An application that performed the same functionality locally on the Android device was also created for the purpose of comparison to the cloud-supported application. It was found that the cloud-supported application completed the task with very little impact on the Android device, and it was capable of much larger workloads than those used during testing. On the other hand, the application that used only local processing required several minutes for a single task and operated at its limits when faced with the tasks performed during testing.

Possible future work includes the development of solutions of a number of Android-related vulnerabilities, including the implementation of tighter controls on access to public files, fixes to the loopholes and inadequacies in the permissions system, and possibly the creation of a data validation framework for Android. A major issue that needs to be addressed is the ability for any application to scan private files for viruses.

# Bibliography

[1] (2009, October) Amazon Usage Estimates. Internet. RightScale. [Online]. Available: http://blog.rightscale.com/2009/10/05/amazon-usage-estimates/

[2] (2011, February) The comScore 2010 Mobile Year in Review. Internet. comScore. [Online]. Available: http://www.comscore.com/Press_Events/Presentations_Whitepapers/2011/2010_Mobile_Year_in_Review

[3] (2011, August) McAfee Q2 2011 Threats Report Shows Significant Growth for Malware on Mobile Platforms. Internet. McAfee. [Online]. Available: http://www.mcafee.com/us/about/news/2011/q3/20110823-01.aspx

[4] Eucalyptus Network Configuration (2.0). Internet. Eucalyptus. [Online]. Available: http://open.eucalyptus.com/wiki/EucalyptusNetworkConfiguration_v2.0

[5] Project Hawaii. Internet. Microsoft. [Online]. Available: http://research.microsoft.com/en-us/um/redmond/projects/hawaii/default.aspx

[6] iCloud. Internet. Apple. [Online]. Available: http://www.apple.com/icloud/

[7] Google Goggles. Internet. Google Mobile. [Online]. Available: http://www.google.com/mobile/goggles/#text

[8] (2011, September) Introducing Amazon Silk. Internet. Amazon. [Online]. Available: http://amazonsilk.wordpress.com/2011/09/28/introducing-amazon-silk/

[9] A. Russakovskii. (2011, October) Massive Security Vulnerability In HTC Android Devices (EVO 3D, 4G, Thunderbolt, Others) Exposes Phone Numbers, GPS, SMS, Emails Addresses, Much More. Internet. Android Police. [Online]. Available: http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more/

[10] J. Case. (2011, April) [Updated] Exclusive: Vulnerability In Skype For Android Is Exposing Your Name, Phone Number, Chat Logs, And A Lot More. Internet. Android Police. [Online]. Available: http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/

[11] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*. IEEE, Dec. 2009, pp. 340–349. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5380692

[12] W. Enck, M. Ongtang, and P. Mcdaniel, "Mitigating Android Software Misuse Before It Happens," Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0094-2008, Nov. 2008. [Online]. Available: http://www.enck.org/papers-ct.html#TechReports

[13] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 347–356. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 301–314. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966473

[15] Y. Chen, V. Paxson, and R. H. Katz, "What's New About Cloud Computing Security?" EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-5, Jan. 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.html

[16] M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono, "On Technical Security Issues in Cloud Computing," in *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, no. 2009. IEEE, Sep. 2009, pp. 109–116. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5284165

[17] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android," in *Proceedings of the 13th Information Security Conference (ISC 2010)*, Oct. 2010.

[18] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google Android: A Comprehensive Security Assessment," *IEEE Security & Privacy Magazine*, vol. 8, no. 2, pp. 35–44, Mar. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5396322

[19] Security and Permissions. Internet. Android Developers. [Online]. Available: http://developer.android.com/guide/topics/security/security.html

[20] Intents and Intent Filters. Internet. Android Developers. [Online]. Available: http://developer.android.com/guide/topics/intents/intents-filters.html

[21] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static Analysis of Executables for Collaborative Malware Detection on Android," in *Proceedings of the 2009 IEEE International Conference on Communications*. IEEE, Jun. 2009, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5199486

[22] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A Small But Non-negligible Flaw in the Android Permission Scheme," in *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 107–110.

[23] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, Jun. 1980. [Online]. Available: http://doi.acm.org/10.1145/358876.358882

[24] Object Detection. Internet. OpenCV. [Online]. Available: http://opencv.willowgarage.com/documentation/cpp/imgproc_object_detection.html

[25] OpenCV Wiki. Internet. OpenCV. [Online]. Available: http://opencv.willowgarage.com/wiki/

[26] W. Simpson. (1996, August) PPP Challenge Handshake Authentication Protocol (CHAP). RFC 1994. [Online]. Available: https://tools.ietf.org/html/rfc1994

[27] The Open Source Toolkit for SSL/TLS. Internet. OpenSSL. [Online]. Available: http://www.openssl.org/

[28] javax.crypto. Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/javax/crypto/package-summary.html

[29] EVP_BytesToKey(3). Internet. OpenSSL. [Online]. Available: http://www.openssl.org/docs/crypto/EVP_BytesToKey.html #KEY_DERIVATION_ALGORITHM

[30] R. Barbosa, "Monitoring local progress with watchdog timers deduced from global properties," in *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, November 2010, pp. 131–140.

[31] H. Rom, "Robust detection & recovery from service disruptions in distributed systems," 2007.

[32] Context.openFileOutput(java.lang.String, int). Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/content/Context.html #openFileOutput(java.lang.String, int)

[33] A. Biryukov and D. Khovratovich, "Related-key cryptanalysis of the full aes-192 and aes-256," Cryptology ePrint Archive, Report 2009/317, 2009, http://eprint.iacr.org/.

[34] E. Jochemsz, "Cryptanalysis of rsa variants using small roots of polynomials," Ph.D. dissertation, Technische Universiteit Eindhoven, 2007.

[35] Activity.onActivityResult(int, int, android.content.Intent). Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html#onActivityResult(int, int, android.content.Intent)

[36] Intent.getData(). Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/content/Intent.html#getData()

[37] Intent.getStringExtra(java.lang.String). Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/content/Intent.html #getStringExtra(java.lang.String)

[38] BitmapFactory.decodeFile(java.lang.String). Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/graphics/BitmapFactory.html #decodeFile(java.lang.String)

[39] H. K. Lee, T. Malkin, and E. Nahum, "Cryptographic strength of ssl/tls servers: Current and recent practices," in *Proceedings of the 7th ACM SIGCOMM Conference*

*on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 83–92. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298318

[40] M. Taufer, D. Anderson, P. Cicotti, and C. B. III, "Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing," in *Proceedings of the 2005 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[41] Preparing to Publish: A Checklist. Internet. Android Developers. [Online]. Available: http://developer.android.com/guide/publishing/preparing.html

[42] Runtime. Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/java/lang/Runtime.html

[43] Creating Toast Notifications. Internet. Android Developers. [Online]. Available: http://developer.android.com/guide/topics/ui/notifiers/toasts.html

[44] VM Control. Internet. Eucalyptus. [Online]. Available: http://open.eucalyptus.com/wiki/Euca2oolsVmControl

[45] ec2-describe-instances. Internet. Amazon Web Services. [Online]. Available: http://docs.amazonwebservices.com/AWSEC2/2007-08-29/DeveloperGuide/CLTRG-describe-instances.html

[46] J. Burns, "Developing Secure Mobile Applications for Android," p. 15, October 2008.

[47] InputFilter. Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/text/InputFilter.html

[48] android:inputType. Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/widget/TextView.html #attr_android:inputType

[49] Debug.MemoryInfo. Internet. Android Developers. [Online]. Available: http://developer.android.com/reference/android/os/Debug.MemoryInfo.html

[50] Android 2.3.3 Platform. Internet. Android Developers. [Online]. Available: http://developer.android.com/sdk/android-2.3.3.html

[51] Getting Started with the AWS SDK for Android (0.2.1). Internet. Amazon Web Services. [Online]. Available: http://aws.amazon.com/articles/4225549089557252

[52] Eucalyptus Community. Internet. Eucalyptus. [Online]. Available: http://open.eucalyptus.com/

# Appendix A

# Image retrieval source

```
/*
 * templates.cpp
 * Image retrieval code
 * Author: Corey A. Beres (cab3674@rit.edu)
 */

#ifdef OPENCV2

#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"

#else

#include "opencv/cv.h"
#include "opencv/highgui.h"

#endif

#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>

#include "Server.h"

using namespace std;
#ifdef OPENCV2
using namespace cv;
#endif
```

```
#define MAX_BYTES 256
char myBestImage[MAX_BYTES];

// Sets the path of the best image to the given char*
void getBestImagePath(char *bestImage, int maxCopyBytes) {
    int stringSize = MAX_BYTES - 1;

    if (maxCopyBytes < stringSize) {
        stringSize = maxCopyBytes - 1;
    }

    strncpy(bestImage, myBestImage, stringSize);
    bestImage[stringSize] = '\0';
}

#ifdef OPENCV2

// Performs template matching on the images in the targetFolder to the sourceImage
// OpenCV 2 version
float getBestMatch(const char *sourceImage, const char *targetFolder) {
    // Source image
    IplImage *img_src = cvLoadImage(sourceImage, CV_LOAD_IMAGE_GRAYSCALE);
    Mat source(img_src, false);
    // Current target image
    IplImage *img_targ;
    Mat target;
    // For results
    float bestScore = -2;   // Initialize to -2 (less than min score of -1.0)
    float tempScore;
    Mat result;
    // File stuff
    DIR *dp;
    char homeDir[256];
    getcwd(homeDir, 256);
    struct dirent *entry;
    struct stat statbuf;

    // Iterate through images in specified folder
    if ((dp = opendir(targetFolder)) == NULL) {
            fprintf(stderr, "cannot open directory: %s\n", targetFolder);
    }
    chdir(targetFolder);
```

```
    while ((entry = readdir(dp)) != NULL) {
            lstat(entry->d_name, &statbuf);
            if (!S_ISDIR(statbuf.st_mode)) {
                    // File, not a directory
                    // Open image and resize it to source size
            img_targ = cvLoadImage(entry->d_name, CV_LOAD_IMAGE_GRAYSCALE);
            Mat target_orig(img_targ, false);
            resize(target_orig, target, source.size(), 0, 0, INTER_CUBIC);
            // Release resources
            target_orig.release();
            cvReleaseImage(&img_targ);

            // Compare it to the source image
            matchTemplate(target, source, result, CV_TM_CCOEFF_NORMED);
            tempScore = result.at<float>(0, 0);
            printf("Best score was %f; %s score is %f\n", bestScore,
                entry->d_name, result.at<float>(0, 0));

            // Check comparison results
            if (tempScore > bestScore) {
                bestScore = tempScore;
                sprintf(myBestImage, "%s/%s", targetFolder, entry->d_name);
            }
            }
    }

    // Release matrix resources
    source.release();
    target.release();
    result.release();
    // Release the image resources
    cvReleaseImage(&img_src);

    closedir(dp);
    chdir(homeDir); // back to our initial directory

    return bestScore;
}

#else

// Performs template matching on the images in the targetFolder to the sourceImage
// OpenCV 1 version
```

```
float getBestMatch(const char *sourceImage, const char *targetFolder) {
    // Source image
    IplImage *img_src;
    // Current target image
    IplImage *img_targ_orig, *img_targ;
    // For results
    float bestScore = -2;   // Initialize to -2 (less than min score of -1.0)
    CvScalar tempScore;
    IplImage *result;
    DIR *dp;
    char homeDir[256];
    getcwd(homeDir, 256);
    struct dirent *entry;
    struct stat statbuf;

    img_src = cvLoadImage(sourceImage, CV_LOAD_IMAGE_GRAYSCALE);
    img_targ = cvLoadImage(sourceImage, CV_LOAD_IMAGE_GRAYSCALE);
    result = cvCreateImage(cvSize(1,1), 32, 1);

    // Iterate through images in specified folder
    if ((dp = opendir(targetFolder)) == NULL) {
            fprintf(stderr, "cannot open directory: %s\n", targetFolder);
    }
    chdir(targetFolder);
    while ((entry = readdir(dp)) != NULL) {
        lstat(entry->d_name, &statbuf);
        if (!S_ISDIR(statbuf.st_mode)) {
            // File, not a directory
            // Open image and resize it to source size
            img_targ_orig = cvLoadImage(entry->d_name, CV_LOAD_IMAGE_GRAYSCALE);
            cvResize(img_targ_orig, img_targ, CV_INTER_CUBIC);
            // Release image
            cvReleaseImage(&img_targ_orig);

            // Compare it to the source image
            cvMatchTemplate(img_src, img_targ, result, CV_TM_CCOEFF_NORMED);
            tempScore = cvGet2D(result, 0, 0);
            printf("Best score was %f; %s score is %f\n", bestScore,
                entry->d_name, tempScore.val[0]);

            // Check comparison results
            if (tempScore.val[0] > bestScore) {
                bestScore = tempScore.val[0];
```

```
                sprintf(myBestImage, "%s/%s", targetFolder, entry->d_name);
            }
            }
    }

    // Release image resources
    cvReleaseImage(&img_src);
    cvReleaseImage(&img_targ);
    cvReleaseImage(&result);

    closedir(dp);
    chdir(homeDir); // back to our initial directory

    return bestScore;
}

#endif
```

# Appendix B

# Cloud worker client source

```
/*
 * Client.cpp
 * Cloud worker client code
 * Author: Corey A. Beres (cab3674@rit.edu)
 */

#include "Client.h"

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <time.h>

// Sockets
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "templates.h"
#include "inputs.h"
#include "Actions.h"

#define PORT 6464

#define BUFFER_SIZE 1024

#define CONNECT_RETRY_DELAY 2

using namespace std;
```

```
int Client::createSocket() {
    socklen_t len;
    struct sockaddr_in address;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr(ipAddr);
    address.sin_port = htons(PORT);
    len = sizeof(address);

    while (connect(sockfd, (struct sockaddr*)&address, len) == -1) {
        printf("Could not connect to server.\n");
        sleep(CONNECT_RETRY_DELAY);
    }

    return sockfd;
}

bool Client::waitForCommand() {
    int actionCode = 0;
    int userId = -1;

    try {
        transmit.recv(&actionCode, sizeof(actionCode));
        printf("Received action code: %d.\n", actionCode);

        switch (actionCode) {
            case ACTION_RAY_TRACE:
                transmit.recv(&userId, sizeof(userId));
                exampleRayTraceScore(userId);
                break;
            case ACTION_BEST_IMAGE:
                transmit.recv(&userId, sizeof(userId));
                exampleRayTraceImage(userId);
                break;
            case ACTION_STATUS_UPDATE:
                transmit.recv(&userId, sizeof(userId));
                updateServer(userId);
                break;
            case ACTION_ECHO:
                transmit.recv(&userId, sizeof(userId));
                echo(userId);
```

```
                break;
            default:
                printf("Received unknown code: %d.\n", actionCode);
                return false;
        }
    } catch(TransmissionException &e) {
        printf("%s\n", e.what());
        if (e.isDisconnectNeeded()) {
            // Return false if we should disconnect
            return false;
        }
    }

    return true;
}

void Client::exampleRayTraceScore(const int userId) {
    char *bestImage = new char[BUFFER_SIZE];
    char *rtCommand = new char[BUFFER_SIZE];
    int code = ACTION_RECV_FLOAT;
    float bestScore;
    time_t rtStart, rtStop;
    double irTime;

    // Perform processing
    sprintf(rtCommand, "/usr/local/bin/rt -st -l 100 -w %d -h %d -f %s",
            INPUT_WIDTH, INPUT_HEIGHT, INPUT_FILE);
    time(&rtStart);
    // Ray tracing
    system(rtCommand);
    time(&rtStop);
    irTime = clock();
    // Image retrieval
    bestScore = getBestMatch(INPUT_FILE, INPUT_FOLDER);
    irTime = (clock() - irTime) / CLOCKS_PER_SEC;
    getBestImagePath(bestImage, BUFFER_SIZE);

    // Save best result for later
    sem_wait(&resultSem);
    results.insert(pair<int, string>(userId, bestImage));
    sem_post(&resultSem);

    // Send best score
```

```
    transmit.send(&code, sizeof(code));
    transmit.send(&userId, sizeof(userId));
    transmit.send(&bestScore, sizeof(bestScore));

    printf("Ray tracing completed in %05.3lf s.\nImage retrieval completed in %05.3lf s.\n",
           difftime(rtStop, rtStart), irTime);
    printf("Sent score %lf to server.\n", bestScore);

    delete bestImage;
    delete rtCommand;
}

void Client::exampleRayTraceImage(const int userId) {
    int code = ACTION_RECV_FILE;
    map<int, string>::iterator iter;

    // Get iterator to result
    sem_wait(&resultSem);
    iter = results.find(userId);
    sem_post(&resultSem);

    // Return result if there is one
    if (iter != results.end()) {
        transmit.send(&code, sizeof(code));
        transmit.send(&userId, sizeof(userId));
        transmit.sendFile(iter->second.c_str());

        // Remove result (otherwise we'll accumulate tons of them)
        sem_wait(&resultSem);
        results.erase(iter);
        sem_post(&resultSem);

        printf("Sent %s to server.\n", iter->second.c_str());
    }
}

void Client::updateServer(const int userId) {
    int code = ACTION_RECV_INT;
    int value = 1;

    transmit.send(&code, sizeof(code));
    transmit.send(&userId, sizeof(userId));
    transmit.send(&value, sizeof(value));
```

```
}

void Client::echo(const int userId) {
    int code = ACTION_RECV_BLOB;
    char *echo = new char[BUFFER_SIZE];
    char *ohce, *pohce;
    int size = 0;
    int newSize = 0;
    int numCopies;
    int i;

    // Receive data size and data
    transmit.recv(&size, sizeof(size));
    if (size > BUFFER_SIZE) {
        size = BUFFER_SIZE;
    }
    transmit.recv(echo, size);

    // Number of times to copy string
    numCopies = BUFFER_SIZE / size;

    // Allocate memory
    ohce = new char[BUFFER_SIZE];
    pohce = ohce;

    // Perform copying
    for (i = 0; i < numCopies; ++i) {
        if (i % 2 == 0) {
            // Copy
            memcpy(pohce, echo, size - 1);
            newSize += size - 1;
            pohce += size - 1;
        } else {
            // Reverse and copy
            reverseAndCopy(pohce, echo, size - 1);
            newSize += size - 1;
            pohce += size - 1;
        }
    }
    // Copy last char (null char?)
    *pohce = echo[size - 1];
    ++newSize;
```

```
    // Send back string
    transmit.send(&code, sizeof(code));
    transmit.send(&userId, sizeof(userId));
    transmit.send(&newSize, sizeof(newSize));
    transmit.send(ohce, newSize);

    delete echo;
    delete ohce;
}

void Client::Run() {
    char *buffer = new char[BUFFER_SIZE];
    /* CHAP stuff */
    CHAP_HEADER chapHeader;
    char *message, *value;
    string sChallenge;
    int msgLen;
    unsigned char chapIdC, chapIdS, valueSize;
    bool valid = false;
    bool success = false;

    printf("Connecting to server...\n");
    createSocket();
    printf("Server connected.\n");

    // Set socket
    transmit.setSockfd(sockfd);

    // Use default key for encryption
    Transmission tempTrans(sockfd);

    try {
        // Receive challenge
        tempTrans.recv(&chapHeader, sizeof(chapHeader));
        if (chapHeader.action == ACTION_CHAP) {
            // Received challenge
            tempTrans.recv(buffer, chapHeader.length);
            valid = chap.challengeBreakout(buffer, chapIdC, value, valueSize);

            if (valid && valueSize > 0) {
                    sChallenge = string(value, valueSize);
                // Make response
                msgLen = chap.response(message, chapIdC, sChallenge.c_str(),
```

```
                                        sChallenge.size(), hash, strlen(hash),
                                        name, strlen(name));
                chapHeader.action = ACTION_CHAP;
                chapHeader.length = msgLen;
                // Send response
                tempTrans.send(&chapHeader, sizeof(chapHeader));
                tempTrans.send(message, msgLen);
                delete message;

                // Receive success/failure
                tempTrans.recv(&chapHeader, sizeof(chapHeader));
                if (chapHeader.action == ACTION_CHAP) {
                    // Received success/failure
                    tempTrans.recv(buffer, chapHeader.length);
                    success = chap.successBreakout(buffer, chapIdS);
                }
            }
        }
    } catch(TransmissionException &e) {
        printf("%s\n", e.what());
    }

    delete buffer;

    if (success) {
        // Authentication succeeded
        for (;;) {
            if (!waitForCommand()) {
                // Disconnect and close
                break;
            }
        }
    }

    pthread_exit(0);
}

void Client::reverseAndCopy(char *out, const char *in, const int num) {
    int c;

    for (c = 0; c < num; ++c) {
        out[num - c - 1] = in[c];
    }
```

```
}

void Client::dispose() {
    int code = 0;
    transmit.send(&code, sizeof(code));
}

/*****************************************************************
 * Constructor
 *****************************************************************/
Client::Client(char *_ipAddr, const char *_name, const char *_hash,
               const char *key) : transmit(key) {
    ipAddr = _ipAddr;
    name = _name;
    hash = _hash;
    sem_init(&resultSem, 0, 1);

    Start();
}

/*****************************************************************
 * Deconstructor
 *****************************************************************/
Client::~Client() {
    close(sockfd);
    sem_destroy(&resultSem);
}

/*****************************************************************
 * Start - creates the thread for the object to run in
 *****************************************************************/
void Client::Start() {
        pthread_create(&tid, 0, StartMe, this);
}

/*****************************************************************
 * StartMe - responsible for starting the control flow of the thread
 * after that thread has been created
 *****************************************************************/
void* Client::StartMe(void* ptr) {
        Client* me = (Client*)ptr;
        me->Run();
        return NULL;
```

```
}

/*****************************************************************
 * GetTid - returns the thread ID
 *****************************************************************/
pthread_t Client::GetTid() {
        return this->tid;
}
```

# Appendix C

# Result manager source

```
package edu.rit.ce.netip.cloud.android.cloudsmart.server;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class RedundantResultManager<T extends Comparable<T>> implements ResultManager<T> {

        private final double RESULT_RATIO = 0.5;

        private ConcurrentHashMap<Integer, T> results;
        private ConcurrentHashMap<T, Integer> resultFreq;
        private int nodes;
        private boolean asked;

        public RedundantResultManager() {
                resultFreq = new ConcurrentHashMap<T, Integer>();
                results = new ConcurrentHashMap<Integer, T>();
                nodes = 0;
                asked = false;
        }

        @Override
        public synchronized int getNodes() {
                return nodes;
        }

        @Override
        public synchronized void setNodes(int nodes) {
                this.nodes = nodes;
        }

        @Override
```

```
public synchronized void clear() {
        resultFreq.clear();
        results.clear();
        nodes = 0;
        asked = false;
}

@Override
public synchronized void removeNode(int nodeId) {
        --nodes;
        results.remove(nodeId);
        asked = false;

        resultFreq.clear();
        for (Map.Entry<Integer, T> result : results.entrySet()) {
                addResult(result.getValue());
        }
}

@Override
public int getBestNode() throws ResultNotReadyException {
        boolean ready = true;
        int nodeId = 0;
        T best = null;
        Map.Entry<T, Integer> mostFreq = getMostFrequent();

        if (nodes < -1) {
                // Must be first
                nodeId = -1;
        } else if (mostFreq.getValue() > RESULT_RATIO * nodes && !asked) {
                // Get mode frequent result
                nodeId = getKeyByValue(mostFreq.getKey());
                asked = true;
        } else if (results.size() >= nodes && !asked) {
                // Have all results, just get the highest result
                for (Map.Entry<Integer, T> result : results.entrySet()) {
                        if (best == null || result.getValue().compareTo(best) > 0) {
                                best = result.getValue();
                                nodeId = result.getKey();
                                asked = true;
                        }
                }
        } else {
```

```
                ready = false;
        }

        if (!ready) {
                throw new ResultNotReadyException();
        }

        return nodeId;
}

@Override
public synchronized void giveResult(int nodeId, T result) {
        results.put(nodeId, result);
        addResult(result);
}

private void addResult(T result) {
        Integer IFreq = resultFreq.get(result);
        int freq = 1;

        // Increment value if it was already present
        if (IFreq != null) {
                freq = IFreq + 1;
        }

        resultFreq.put(result, freq);
}

private Map.Entry<T, Integer> getMostFrequent() {
        Map.Entry<T, Integer> freq = null;

        for (Map.Entry<T, Integer> rf : resultFreq.entrySet()) {
                if (freq == null || rf.getValue() > freq.getValue()) {
                        freq = rf;
                }
        }

        return freq;
}

private int getKeyByValue(T value) {
        int key = 0;
```

```
                for (Map.Entry<Integer, T> result : results.entrySet()) {
                        if (result.getValue().equals(value)) {
                                key = result.getKey();
                        }
                }

                return key;
        }
}
```

# Appendix D

# Android application, main loop source

```
public void run() {
    byte[] buffer, challenge, header;
    int byteRead = 0;
    float score = 0;
    boolean done = false;
    boolean success = false;
    int length;
    byte chapId;
    Aes tempAes = new Aes();

    while (!done) {
        try {
            connectToServer(host, PORT);
            done = true;

            try {
                // Perform CHAP authentication
                buffer = receiveEnc(bis, tempAes, CHAP_HEADER_SIZE);
                if (byteArrayToInt(buffer) == ACTION_CHAP) {
                    // Receive challenge
                    length = CloudClient.this.getChapMessageSize(buffer);
                    buffer = receiveEnc(bis, tempAes, length);
                    chapId = chap.getId(buffer);
                    challenge = chap.getChallenge(buffer);

                    // Make/send response
                    if (usePap) {
                        // PAP
                        buffer = getPapMessage();
                        header = getPapHeader(buffer.length);
                        sendEnc(bos, tempAes, header);
                        // Send as plain text, just to prove a point
```

```
            bos.write(buffer, 0, buffer.length);
            bos.flush();
        } else {
            // CHAP
            buffer = chap.response(chapId, challenge, hash.getBytes("UTF-8"),
                    username.getBytes("UTF-8"));
            header = getChapHeader(buffer.length);
            sendEnc(bos, tempAes, header);
            sendEnc(bos, tempAes, buffer);
        }

        // Receive result
        buffer = receiveEnc(bis, tempAes, CHAP_HEADER_SIZE);
        if (byteArrayToInt(buffer) == ACTION_CHAP) {
            length = CloudClient.this.getChapMessageSize(buffer);
            buffer = receiveEnc(bis, tempAes, length);
            chapId = chap.getId(buffer);
            success = chap.getSuccess(buffer);
        }
    }

    if (success) {    // Authentication was successful
        // Ask server for result
        if (chooseResult) {
            // Get intermediate results from servers
            sendEnc(bos, intToByteArray(ACTION_BEST_SCORE));
        } else if (multipleClients) {
            // Make server use multiple clients
            sendEnc(bos, intToByteArray(ACTION_RAY_TRACE));
        } else {
            // Make server use a single client
            sendEnc(bos, intToByteArray(ACTION_SINGLE_RAY_TRACE));
        }

        if (chooseResult) {
            // Begin receiving data
            buffer = receiveEnc(bis, 4);
            if (buffer != null) {
                byteRead = byteArrayToInt(buffer);
                Log.i(TAG, "Received byte " + byteRead);
            } else {
                byteRead = -1;
            }
```

```
                        if (byteRead == ACTION_FLOAT_RESULT) {
                            buffer = receiveEnc(bis, 4);
                            if (buffer != null) {
                                score = Float.intBitsToFloat(byteArrayToInt(buffer));
                                rm.giveResult(id, score);
                                setChanged();
                                notifyObservers();
                                Log.i(TAG, "Received score " + score);
                            } else {
                                byteRead = -1;
                            }
                        }
                    } else {
                        // File will be sent automatically
                        rm.giveResult(id, 1f);
                        setChanged();
                        notifyObservers();
                    }
                }
            } catch (IOException e) {
                Log.e(TAG, e.getMessage());
            }
        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```